University of Windsor
Scholarship at UWindsor

Electronic Theses and Dissertations

2001

Experienced agents with attitude in a virtual marketplace.

Pratapa Reddy. Sathi
University of Windsor

Follow this and additional works at: http://scholar.uwindsor.ca/etd

Recommended Citation
Sathi, Pratapa Reddy., "Experienced agents with attitude in a virtual marketplace." (2001). *Electronic Theses and Dissertations.* Paper 2014.

This online database contains the full-text of PhD dissertations and Masters' theses of University of Windsor students from 1954 forward. These documents are made available for personal study and research purposes only, in accordance with the Canadian Copyright Act and the Creative Commons license—CC BY-NC-ND (Attribution, Non-Commercial, No Derivative Works). Under this license, works must always be attributed to the copyright holder (original author), cannot be used for any commercial purposes, and may not be altered. Any other use would require the permission of the copyright holder. Students may inquire about withdrawing their dissertation and/or thesis from this database. For additional inquiries, please contact the repository administrator via email (scholarship@uwindsor.ca) or by telephone at 519-253-3000ext. 3208.

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI®

Experienced Agents with Attitude in a Virtual Marketplace

by

Pratap R. Sathi

A Thesis
Submitted to the Faculty of Graduate Studies and Research
through School of Computer Science in Partial
Fulfillment of the Requirements for the Degree of
Master of Science at the
University of Windsor

Windsor, Ontario Canada
2001

National Library of Canada

Acquisitions and Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale du Canada

Acquisitions et services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

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

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

0-612-62280-0

Canadä

Pratap R. Sathi

© All Rights Reserved

Abstract

Agent technology evolved from a number of disciplines such as object technology, distributed computing and artificial intelligence. There is currently a growing need to develop intelligent agents that engage in buying and selling in virtual marketplaces owing to exponential growth of e-commerce. For e-commerce to become a reality, and before users can trust software agents to buy and sell items on their behalf, some "intelligence" that makes humans good buyers and sellers must be simulated. In this thesis report we suggest a prototype of a virtual marketplace where agents negotiate to buy and sell goods. The agents in this framework represent the three important attributes of mental state (attitude) of their clients. These agents learn limitedly from their previous experiences and adjust their attitude. Learning from experience is modeled using Case Based Reasoning (CBR) Techniques. Our proposed framework suggests that it is possible to build software agents, which model limited aspects of negotiation in buying and selling.

To My Parents Anu Radha & Krishna Reddy

and three people who influenced the most crucial phase of my life (last six years) filling me with Confidence, Advices, and Motivation:

Dr. Walid S. Saba
Dr. Krishna P. Reddy
Mr. Umesh B. Sidhra

Acknowledgments

I am indebted to my advisor Dr. Walid S. Saba for introducing me to intelligent agents, for his valuable guidance and advices, and for the motivation and support provided all along. I specially thank my committee members Dr. Richard A. Frost, Dr. Yash P. Aneja and Dr. Alioune Ngom for their valuable suggestions and support. I thank Dr. Ahmad Tawfik and other faculty members for their valuable suggestions and feed back. I also thank Mr. Nanth Kumar, Mr. Vijay Kallepalli, and other friends who helped me through out this work. My thanks are due to SAINT seminars and members of LEGroup for their insightful comments about this work.

Contents

Abstract iv

Dedication v

Acknowledgements vi

List of Figures xi

List of Tables xii

1 Introduction 1

1.1 Software Agents in Virtual Marketplaces 1
1.2 Rationale for Automated Negotiation 2
1.3 The Problem Domain 3
1.3.1 Problems in E-commerce Negotiation 3
1.3.2 Thesis Contribution 5
1.4 Outline of the Proposed Framework 5
1.5 Structure of the Thesis 6

2 An Overview of Existing Approaches 7
2.1 Kasbah 7
2.1.1 Kasbah Architecture 7
2.1.2 Agent Behaviour in Kasbah 8
2.1.3 Evaluation of Kasbah 8
2.2 Experience Based Learning 9
2.2.1 Negotiating with experience 9
2.2.1.1 Architecture of Experience Based Negotiator 10
2.2.1.2 Agent Behaviour in Experience Based Negotiator 10
2.2.1.3 Evaluation of Experience Based Negotiator 11
2.3 Similarity Criteria in Negotiation Trade-offs 11
2.3.1 Overview of Negotiation process 11
2.3.2 Agent Behaviour 12
2.3.3 Evaluation of the approach 12
2.4 Multi-Agent based Learning Economy (MAGALE) 13
2.4.1 Overview of negotiation process 13
2.4.2 Agent Behaviour 14
2.4.3 Evaluation of the Model 14
2.5 Negotiation Model for Multiple Transaction Factors in E-commerce 15
2.5.1 Overview of negotiation process 15
2.5.2 Agent Behaviour in the system 15
2.5.3 Evaluation of the Model 16
2.6 Economic Models 16

2.7 Epistemic Logics 17
2.8 Assessment of Existing Approaches 17

3 Our Framework for Agent Negotiation in a Virtual Marketplace 19
3.1 Negotiation Protocol 19
3.2 Negotiation Issues 19
3.3 Overview of the Virtual Marketplace 20
3.3.1 Basic Components of the Framework 20
3.3.2 Overview of the Negotiation Process 21
3.4 Definitions used in this Framework 22
3.4.1 Agents Attitude 22
3.4.2 Public Price Range (PPR) 23
3.4.3 Supply/Demand Ratio 23
3.4.4 Negotiation 24
3.4.5 Agent Experience 24
3.4.6 Agent's Price Range (APR) 25
3.5 The Negotiation Process in detail 28
3.6 Case-Based Reasoning and Learning from Experience 29
3.6.1 Matching Cases 31
3.6.1.1 Product Similarity (PS) 32
3.6.1.2 Attitude Similarity (AS) 35
3.6.1.3 Public Price Range (RS) 36
3.6.1.4 Matching most Relevant Case 37
3.6.2 Learning from Experience 38
3.6.2.1 Adjusting Attitude 38
3.6.2.2 Calculating Agent's Price Range 41
3.6.2.3 Calculating Bid Increment α 41
3.6.3 Saving Experience for Future Use 42
3.6.3.1 Negotiation Similarity (NS) 43
3.6.3.2 Updating Casebase 44
3.7 Conclusion 45

4 Implementation Details 46
4.1 Building the Environment 46
4.2 Object Diagram 47
4.3 Implementing the Process of Negotiation 48
4.3.1 Creation of Agents and Clone 49
4.3.2 Retrieving Similar Products from Ontology 50
4.3.3 Retrieval and Adaptation of Relevant Experience 52
4.3.3.1 Representing Cases (Experience) 52
4.3.3.2 Choosing Relevant Experience 53
4.3.3.3 Adapting a Case 53
4.3.3.4 Updating Casebase 53
4.3.4 Implementation of Similarity and Other Functions 54

5	**Implementation Details**	**55**
	5.1 Implementation of Environment	55
	5.2 Implementation of Ontology	56
	5.3 Implementing Retrieval of Relevant Experience	56
	5.4 Implementing the functionality of Buyer and Seller Agents	57
	5.4.1 Functionality of Buyer	57
	5.4.2 Functionality of Seller	57
	5.5 Implementing functionality of Clones	58
	5.5.1 Functionality of Buyer Clone	58
	5.5.2 Functionality of Seller Clone	58
	5.6 Challenges in Implementation	58
6.	**Evaluation of the Proposed Framework**	**61**
	6.1 Behaviour of Agents in the Framework	62
	6.2 Scenarios considered for Experiments	64
	6.2.1 Scenarios representing various Experiences	65
	6.2.2 Scenarios representing various Degree of Interactions	65
	6.3 Experiments and Results	66
	6.3.1 Experiments testing scenarios with various experiences	66
	6.3.2 Experiments with various degrees of interaction	67
	6.4 Computational Issues	69
	6.4.1 Computational Efficiency	69
	6.4.2 Communication Efficiency	71
	6.4.3 Distribution of Computation	71
	6.4 Critical Analysis of the Model	72
	6.4.1 Limitations	72
	6.4.2 Compromises	73
	6.4.3 Scalability Issues	74
7	**Conclusion and Future Work**	**76**
	7.1 Framework to Automate Negotiations	76
	7.2 Analysis of Experiments	77
	7.3 Future work	78
8	**Bibliography**	**80**
		80
A	**Software Agent Technology**	**84**
	A.1 Software Agent Technology	84
	A.2 What is an Agent?	85
	A.3 Common Properties	87
	A.4 Classifying Properties	89
	A.5 Classification of Agents	90

B	**Negotiation**	**95**
	B.1 Negotiation and Negotiation Theory	95
	B.2 Negotiation Process	96
	B.3 Automated Negotiation	97
C	**Case Based Reasoning**	**99**
	C.1 Case Representation	100
	C.2 Case Indexing	100
	C.3 Case Retrieval	101
	C.4 Case Adaptation	101
	C.5 Case Evaluation	101
	C.6 Case Retainment	101
D	**Results**	**103**
	D.1 Experiments using an Unsuccessful Experience to buy similar product	103
	D.2 Merging Cases	106
	D.3 Many-many Interactions	108
E	**Documented Code**	**111**
	Vita Auctoris	**160**

List of Figures

Figure 2.1	Price adjustment curves for selling agents in Kasbah	8
Figure 3.1	Basic Components of Virtual Marketplace	21
Figure 4.1	Basic Components of the Virtual Marketplace	46
Figure 4.2	Object Diagram of the Virtual Marketplace	47
Figure 4.3	Object Interactions in the proposed Framework	48
Figure 4.4	Sample Ontology for Consumer Electronics Domain	50
Figure A.1	Evolution of Agent Technology	85
Figure A.2	Agent typology based on three properties	91

List of Tables

Table 4.1	Database tables representing the Conceptual Hierarchy	51
Table 4.2	Database table representing Cases	52
Table D.1	Buyer Experiences in Database	103
Table D.2	Seller Experiences in Database	104
Table D.3	A new experience of buying a Monitor	106
Table D.4	Updated experience of buying a Monitor	108

Chapter 1

Introduction

1.1 Software agents in Virtual Market places

Over the past few years exponential growth of e-commerce on the Internet motivated many researchers in academia and software industry to focus on importance of software agents in virtual market places. Software agents appeared based on a synthesis of ideas from Artificial Intelligence (AI), Object Oriented Technology (OO) and Distributed Computing. According to [35], Software agents differ from "traditional programs" in that they are autonomous, situated in an environment, proactive and reactive. Software agents can communicate, move from place to place, negotiate for resources and learn. These features led the researchers to increase their interest in software agents to counter the challenges in today's heterogeneous, distributed, and uncertain (dynamic) e-commerce environment.

Current e-commerce environment only supports non-interactive buying-selling and auction type mechanisms. The challenges for e-commerce users include loads of information, rapid change of content, spending time on searching for goods. There are no models yet for automated negotiation in E-commerce. Many existing systems in e-commerce lack evaluation of product features and provisions for negotiations. Other concerns include consumer-buying behaviour, security and trusting a piece of software for major purchases. Most of these challenges can be countered by introducing buying and selling agents, in other words virtual buyers and sellers. Buying and selling agents can consult with Recommender systems [36] for product recommendation, the second

step in consumer buying behaviour. With the recommendations of the Recommender agents, the buying and selling agents visit the virtual markets and negotiate on users behalf.

1.2 Rationale for Automated Negotiation

Negotiation is a form of decision-making where two or more parties jointly search a space of possible solutions with the goal of reaching a consensus for their own benefits. Negotiation theory[1] covers a range of phenomena and encompasses multifarious approaches like Artificial Intelligence, Social Psychology, and Game theory [32]. Negotiation deals with three broad topics [25], viz., negotiation protocols, a set of rules, which govern the negotiation process; negotiation objects, range of issues over which agreements are to be reached; Agents' Decision Making Models, and the decision making apparatus used to achieve negotiation objectives. The relative importance of these three topics may vary according to the negotiation and environmental context. In the process of negotiation the proposer makes a proposal to the recipient and in turn the recipient responds with feedback about the proposal in the form of critic or a counter proposal. To make the proposals and counter proposals the agents should act intelligently and should have some reasoning capabilities. When these agents are equipped with negotiation techniques, tactics, and reasoning capabilities, they can negotiate without any human interaction at any virtual market place like classified negotiation, stock market negotiation and retail auction negotiation.

[1] See appendix for details about negotiation theory

In terms of virtual markets, the negotiation phase is viewed as the process where by buying and selling agents autonomously communicate their proposals and counter proposals to reach a common decision which would benefit both parties namely buyers and sellers. Automated negotiation saves time for people as it reduces the time consuming activities like negotiating for price and other product features. The autonomous, proactive and reactive nature of these agents makes them suitable candidates for automated negotiation in uncertain complex e-commerce transactions. To be competitive in today's dynamic e-commerce environment, the selling and buying agents need to act rationally and intelligently. They should be able to reason and learn for effective negotiation.

1.3 The Problem Domain

1.3.1 Problems in E-commerce negotiation

This subsection presents the problematic issues in automated negotiation, which motivated this thesis and the next subsection elaborates on the contribution of this thesis towards automated negotiation.

Formalization of the above discussed negotiation process is very complex as there are many protocols and properties to be considered [27]. The general properties desirable for a negotiation mechanism are computational efficiency, communication efficiency, distribution of computation, and individual rationality. The former three issues pose major software engineering challenges. Individual rationality is more complex and challenging as it depends on the following parameters

1. Cardinality of negotiation (1 to 1, 1 to many, and many to many)

2. Agent characteristics (role, knowledge, commitment)
3. Environment and goods characteristics (private-public value, numbers)
4. Event parameters (importance of time, schedules)
5. Information parameters (price quotes, transaction history (experience))

These parameters may completely vary from domain to domain. Therefore, in most cases negotiation strategies and tactics are completely domain dependant. In E-commerce set up negotiation gets really complex, as the parameters here are fuzzy, dynamic and very diverse. The challenges for automated negotiation in E-commerce applications include the following [13]:

1. It is very difficult to expect an automated negotiation process that reflects the real world
2. There is no negotiation based on diverse attributes.
3. There is no multi-negotiation that considers and is adapted to all counterparts participating in negotiation process simultaneously
4. There is no personalized negotiation

In summary, involvement of many parameters makes automated negotiation a really complex process and there is no universally accepted negotiation technique. The broader issues to be addressed include, development of good negotiation techniques, which would suit the specific domain, relative strengths and weaknesses of each technique should be thoroughly explored and generation of predictable negotiation behaviours need to be developed. "At present there has been virtually no work on how a user can instruct an agent to negotiate on their behalf" [27].

1.3.2 Thesis Contribution

The thesis that we defend in this report is the following:

"It is possible to build autonomous agent systems which model limited (important) aspects of negotiation using current software technologies"

This thesis proposes a framework for automated negotiation among software agents in e-commerce environment. The agents in this framework tend to learn from their experience over a period of time, on incorporation of different learning algorithms. Learning from experience in these agents is modeled using case based reasoning[2] (CBR). The agents in this framework can adapt themselves to solve the given problem and generate solutions in negotiation process based on their previous experience, world knowledge, and their attitudes. This framework uses ontology to equip these agents with domain specific knowledge for reasoning purposes and postulates interesting functions and rules, which help agents to negotiate autonomously.

1.4 Outline of the Proposed Framework

The framework proposed in this thesis is a prototype of a virtual market place where buying and selling agents autonomously negotiate on behalf of their clients. Though this negotiation framework shares several common features with a number of existing approaches to negotiation [4, 16, 29, 38], this framework uniquely explores the interaction between agent's attitude and it's prior experience. Moreover, the agents in

[2] See appendix for details on Case Based Reasoning

this framework consider environmental factors like supply and demand issues while negotiating.

The work in this thesis is original in the following respects:

1. *"Representing three important attributes of mental state (attitude) of the clients to drive the whole negotiation process".*
2. *"Interaction between mental state and prior experience in the context of agent negotiation".*

Analysis of this framework shows that this virtual marketplace prototype is a promising step towards exploiting the advantages of automated negotiation in today's uncertain, dynamic e-commerce environment.

1.5 Structure of the Thesis

The remainder of the thesis is structured as follows: In chapter 2, a review of related work is discussed. In chapter 3, the process of negotiation, parameters considered, and functions that drive negotiation in this framework are described. Chapter 4 and 5 discuss the implementation issues and implementation details respectively. In chapter 6, analysis of the framework and computational issues are discussed. Chapter 7 discusses the conclusions drawn from the framework and observations regarding future extensions are described.

Chapter 2

An Overview of Existing Approaches

The aim of this chapter is to review related models and approaches for automated negotiation in e-commerce applications. Automated negotiation is a very promising area of research in the wake of exponential growth of e-commerce. The complexity of the process and the parameters involved in the process of negotiation has been attracting researchers for a decade.

Current approaches suggested by researchers in automated negotiation for e-commerce include, predefined and non-adaptive negotiation mechanisms in Kasbah [4], using similarity criteria to make negotiation tradeoffs [29], experienced based negotiation approach [39], and using influence diagrams and decision theoretic tools to predict opponent actions during negotiation [12]. This chapter reviews these approaches briefly and evaluates each approach.

2. 1 KASBAH

Kasbah, created in the MIT Media Laboratory, is an electronic agent marketplace where agents negotiate to buy and sell goods and services on behalf of the user. Kasbah assists users in electronic shopping, as the agents autonomously negotiate and make the deals on the users behalf.

2.1.1 Kasbah Architecture

Kasbah has three major components.

1. *Front-end*: a web interface to handle user interaction.
2. *Back-end*: a market place engine where agents operate and interact with each other.
3. *Auxiliary components*: To generate visual display files, notifications to user.

2.1.2 Agent behavior in Kasbah

Buying and selling agents in Kasbah negotiate based on a predetermined strategy chosen by the user. For example, a selling user defines the goal of the selling agent by specifying desired price, lowest acceptable price, and desired date to sell. There are three pre-determined strategies viz., anxious, cool-headed, and greedy which are linear, quadratic, and cubic decay functions respectively as shown in the following figure [33].

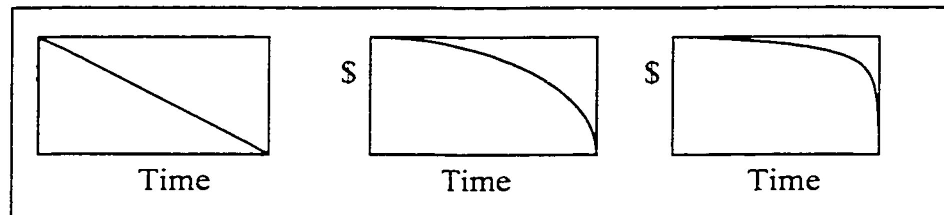


Figure 2.1: Price adjustment curves for selling agents in Kasbah
(Left to right: "anxious", "cool-headed", "greedy")

During the process of negotiation if the other party accepts the desired price, the process is terminated. Otherwise the selling agent lowers the price till it reaches the lowest acceptable, as per one of the above strategies. Intuitively, it works opposite way in case of a buyer.

2.1.3 Evaluation of Kasbah

Kasbah is one of the initial efforts to model real world negotiation as it models time, actions and strategies in negotiation. The agents in Kasbah are simple and there is no

learning in the system. The decisions delegated to agents in Kasbah are only limited to only three (mentioned in above figure) and their selection is not autonomous. Moreover, only one parameter, the price, drives negotiation in Kasbah. Negotiation could be done over multiple parameters, where agents can make concession over one or more issues. In summary, Kasbah does not support negotiation on multiple attributes and there is no learning in the agents in the system.

2.2 Experienced based learning

The Learning from experience approach is based on R. Schank's *Theory of Dynamic memory*, where it is assumed that humans understand by trying to integrate new things we encounter with what we already know (experienced). In the process of understanding we remember old experiences, consciously or unconsciously, as we process the new ones. This theory says remembering, understanding, experiencing and learning cannot be separated from each other. Cognitively, good negotiation skills in humans seem to come from experience. Few researchers in AI used Case Based Reasoning as a learning approach in these buying and selling agents.

2.2.1 Negotiating with experience (Wai Wong, Dong Zhang, Mustapha Ali, 2000)

This system uses Case-Based Reasoning (CBR) as a tool to use past experience and strategies in developing negotiation strategies for current situation. This experience based negotiation framework provides adaptive negotiation strategies that can be generated dynamically and are context-sensitive.

2.2.1.1 Architecture of Experience Based Negotiator

1. *Case-Based Negotiator*: To assist the users to negotiate with opponent agents. It matches current negotiating scenario with previous successful negotiation cases and provides appropriate counter-offers for the user based on the best-matched negotiation case.
2. *Case Browser*: Allows users to browse a previous negotiation case repository using various queries.
3. *Case Maintenance component:* Allows negotiation experts to moderate, maintain and to update the case repository.

2.2.1.2 Agent behavior in Experience Based Negotiator

The agents in this model are assumed to berational. The negotiation process here is strictly monotonic. Initially the case base is populated with number of cases with relevance to the product domain of the application. Once the agent is created it gets the relevant previous cases from the case base then it selects the most matched case. A concession match filter is used to match the cases. The agent then uses that previous experience in generating the solution for the current problem.

In the previous negotiation experiences, a series of offers/counter-offers and concessions reflects information related to episode strategies. The changes from one offer/counter-offer to another offer/counter-offer show variation of episode strategies. If the opponent agent's offer reaches a plateau and offer is within the agent's budget, suggest a concession that can reach the opponent agent's offer. Some adaptation is done during the boundary conditions. If the concession results in a counter-offer, which is higher than the buyer's maximum budget, then the concession must be

adjusted accordingly to arrive only at the counter-offer equal to or not greater than the buyer's maximum budget. If there are no matches in the case base similar to present case then the agents have to rely on some predefined strategies.

2.2.1.3 Evaluation of Experience Based Negotiator

Experience makes humans better negotiators. This work is impressive and is a good milestone towards building a negotiating model, which would represent people's buying behavior. The agents in this system learn from experience. This work is limited to only one particular domain and negotiation is on single attribute, price. Moreover the attitudes of the buyers and sellers are not taken in to consideration. Attitude[3] of the user plays major role in transactions along with his experience. This work mostly concentrated on the matching of cases, with less focus on actual proposal generation and learning from failure is not considered.

2.3 Similarity criteria in negotiation trade offs (P. Faratin, C. Sierra, N. R. Jennings, 2000)

This work is not directly related to virtual market places at this point, but it has significant importance in terms of the strategies used, as they are related to the research in this thesis.

2.3.1 Overview of negotiation process

This work is related to a particular class of negotiation: service oriented negotiation, where a producer and a consumer have to come to a mutually acceptable agreement over terms and conditions under which the producer will execute some problem solving

---

[3] Attitude of the user may include many factors. Some obvious factors in an attitude could be importance of time (Urgency), importance of Price (price consciousness), and Commitment of the user for the given transaction.

activity for the consumer. The issues of concern may include price of the service, time at which it is required, quality of the service, and the penalty to be paid for reneging the agreement.

2.3.2 Agent behavior

Agents may make trade-offs in these negotiations by accepting a service of lower quality if the price is cheaper or accepting a shorter deadline if it receives a higher price. In making a negotiation trade off, an agent tries to find a contract that has the same value to itself as the previous proposal, but more acceptable to its opponent. As the agent does not know the opponent's utility functions, it approximately estimates them using similarity functions. Fuzzy similarity concepts are used to compute similarity between the contracts.

The trade off algorithm in this work performs an iterated hill-climbing search of possible contracts. This search results in generating contracts that lie closer to the iso-curve. This algorithm randomly generates the contracts by splitting the gain in utility, among the set of issues under negotiation. The agent then selects the contract that maximizes the similarity to the opponent's last offering. This algorithm terminates when the last selected contract lies on the iso-curve. Thus the agents in this system generate the contracts so that there are joint gains for both parties in the process of negotiation.

2.3.3 Evaluation of this approach

The issue of negotiation trade-offs has been ignored in automated negotiation. This work substantially contributes towards negotiation trade-offs and multi issue negotiation. The similarity functions in this work addresses the issue of matching

parameters under uncertainty. The trade-off algorithm generates reasonable trade-offs in contracts when there is limited information about the opponents in negotiation. Though this work seems to be impressive in service-oriented negotiation, there is a necessity to consider more parameters when it is applied to a domain like automated negotiations. The attitude of the agents is not taken in to account in this work.

2.4 Multi-Agent based Learning Economy (MAGALE)(C. Mudgal, J. Vassileva, 2000)

This work attempts to create a market place for learning resources (an e-commerce environment for trading intangible goods like advice, help teaching or tutoring).

2.4.1 Overview of negotiation process

There is asynchronous and synchronous information exchange in this system. Asynchronous information includes web pages, FAQ entries, and asking a question via email. Synchronous information includes an online help session via chat, telephone or collaboration environment, where real-time live contact is made. Synchronous information exchange needs negotiation, as the price has to be determined dynamically, since many factors play a role. For example, how urgent the help is needed, and how busy are the helpers in the system. The user who possesses knowledge resources becomes seller and the user who seek the help or advice becomes buyer in this market place. The agents in this system decide how to increase or decrease the price depending on user's preferences.

2.4.2 Agent behavior

The agents in MAGALE represent the users. They maintain information about user's goals, preferences and knowledge. Once the user needs help, the agent gets the information about the opponent users via a matchmaker in respect to the user needs. Agents make decisions on behalf of their users about the price to offer to strike a better deal. Agents make offers and counter offers based on their preferences iteratively. As there is high degree of uncertainty in the state of the market or opponent's preferences, an influence diagram[4] models the uncertain variables and decisions. Evaluating the influence diagram gives an optimal solution for the problem. Moreover, influence diagrams help the agents to predict the opponent's reaction.

The decisions by the agents in this model also driven by the preferences like importance of money, urgency, and risk behavior of the user. For example, risk seeking agent will try to counter propose an offer rather than accepting it, on the other hand a risk averse agent will accept whatever minimum price is offered and refrain form counter proposing in fear of losing the deal. The state of the agent is in any of the following states: Accept, Reject, or Counter propose.

2.4.3 Evaluation of the Model

The use of probabilistic influence diagrams helped the agents to model their opponents and predict their behavior during the negotiation process. Preferences (attitudes) of the user are also considered in agent decision-making during negotiation. But it is still an open question how close the probabilistic diagrams could match the opponent's

---

[4] For more details on Influence Diagrams see [6].

reactions in a dynamic environment. Moreover, the process could be very complex in case of large number of users in the system.

2.5 Negotiation Model for Multiple Transaction Factors and Learning in E-commerce (J. Kang, E. Lee 1998)

This work proposed a negotiation model where agents negotiate for diverse attributes simultaneously with use of a Black Board. The system uses an ontology to represent knowledge and the strategies in this system are environment-adaptive.

2.5.1 Overview of negotiation process

The Customer Agents (CAs) and Supplier (SAs) negotiate to obtain a more profitable deal on behalf of the user's goal. A buyer in this system searches for the right counter parts (Supplier Agents) and then exchange offers and counter offers. During the process of negotiation the CA's notify the Black Board about the proceeding state of the negotiation process. Using the notifications on Black Board the CAs are able to compare and analyze the negotiation strategies of all SAs and are also able to understand and analyze the negotiation trend of all SAs. The agents transmit the message with priority of each attribute to the counterpart. After receiving the message, counterpart may make an offer or counteroffer based on the priority of the counterpart.

2.5.2 Agent behavior in the system

Agents in this system learn and come up with new strategies from time to time. They learn from the information on the Black Board and understand the general trend, which is item specific. This helps the agent to select it's own strategy safely and conveniently. The agents in this system also learn by understanding other counterpart's

strategy and come up with new strategy for counteroffers. Thus the CAs learn the new strategy by comparing and analyzing the strategies of counterparts and also trend variation in negotiation in the environment, which is supplied by the Black Board.

2.5.3 Evaluation of the model

This negotiation process helps the agents to compare and analyze the strategies simultaneously; thereby the agent can come up with new strategies. The learning in this system is rule-based learning, where strategies are made out of observation from the environment and opponents strategies. Creation of replicas for agents makes execution of tasks faster in this system. But, in real time e-commerce negotiations where the negotiation attributes are very dynamic and where agents tend to hide information, the agents in this model would fail to learn from opponent's strategies as they are hidden.

2.6 Economic Models (Game theory)

In game theory an agent is viewed as an individual, a firm or some complex organization. An agent is an optimizer of some function for maximization of profit. Game theory models do not describe *how the world is or must be*, but they describe *how the world could be.*

When rational players play the game, the outcome of the game is decided by the information in the structure of the game. The outcome of the game is determined by singling out an optimal strategy for each agent. Various criteria of individual optimality in game theory include Dominance, Nash Equilibrium, Bayesian Nash Equilibrium, trembling hand equilibrium, and sequential equilibrium. Each equilibrium

concept refines another. Nash equilibrium is best-known strategy for negotiation. This theory predicts a unique solution to each game chosen by the agent. The predicted strategy of each agent must be the agent's best response to the predicted strategies of the other agents and it should maximize the utility. Negotiation systems have been built based on game theoretic models, where agents use these models for bargains on deals [38].

2.7 Epistemic Logics

Distribution and transfer of information among autonomous agents are critical characteristics in many environments. Representing and reasoning about the state of information and dynamics of information is very important to analyze these environments. The formalisms that support such representation and reasoning are called 'epistemic logic' or 'logic of knowledge'.

Epistemic logics have been applied even outside of computer science, especially in economics and game theory. In relation to computer science, epistemic logics are very useful in analyzing distributed systems. State of distributed systems could be characterized using epistemic logics in terms of availability of information to each processor, information needed by the processor and common knowledge among the processors. Dynamics of distributed systems could be characterized in terms of transfer of information among the processors through communication. In building distributed environments for software agents, there is a need to focus on epistemic logics to represent the knowledge about the system.

2.8 Assessment of Existing Approaches

Modeling of human buying behavior involves complex issues like user's experience, many negotiation strategies, many parameters like product attributes, attitude of the users, and minor economic issues like supply and demand ratios.

Brief review of related approaches discussed above give us an understanding that none of them cover all the issues needed to model the human buying behavior.

Both, Kasbah and the Black Board model described in sections 2.1 and 2.5 respectively, lack modeling the use of experience. Kasbah completely depends on few pre-defined strategies and there is no learning in the system.

Similarity criteria and MAGALE described in sections 2.3 and 2.4 models attitudes of the agents and few aspects of learning using similarity functions and influence diagrams respectively. Moreover these two approaches are more suitable to service oriented negotiation than to the problem domain of this research, product-oriented negotiation.

The Experience based Negotiator described in section 2.2 models learning from experience, but did not concentrate on modeling the attitudes of the agents and sophisticated negotiation strategies. Learning from failure is also not considered.

Brief assessment of these approaches shows that in modeling human buying behavior there is a need for a sophisticated framework which could incorporate different strategies for negotiation, focuses on various parameters that affect negotiation, makes use of learning from experience, and uses reasonable functions to drive the negotiation process.

Chapter 3

Our Framework for Agent Negotiation in a Virtual Marketplace

In this chapter we describe a framework for automated negotiation in a virtual marketplace. This prototype specifies the protocol of interaction, various parameters involved in negotiation, negotiation decision-making, and is intended to model issues identified in section 1.3.1 of chapter 1 and also addresses criticisms of the existing approaches in chapter 2.

As explained earlier in chapter 1, the problem domain of this research is automated negotiation in e-commerce applications, in other words virtual marketplaces.

3.1 Negotiation Protocol

Agents need to interact and communicate with other agents in the virtual marketplace. A protocol of interaction is required to model and facilitate the interaction among the agents. The interaction is modeled as alternating sequences of offers and counteroffers, which terminates with a successful negotiation deal, or when both parties fail to reach an agreement on a deal. The interaction among agents in this framework will be explained in more detail in following sections of this chapter.

3.2 Negotiation Issues

Most models of negotiation deal with only a single issue, namely price, as discussed in the previous chapter. Negotiation in our framework depends on various attributes like price, warranty, and the attitude of the user (urgency of the user, importance of price,

level of commitment). Urgency and Commitment of the user could play a major role in negotiation while making offers and counter offers. Thus the negotiation in this framework is based on multiple issues. Other factors that could the affect negotiation in this framework include, environmental factors like supply demand ratio and experience of the user.

3.3 Overview of the virtual market place

The interaction between agents in this marketplace is many-many. Buying and Selling agents negotiate on behalf of their users in the virtual marketplace. This section discusses the basic components and negotiation process of this framework.

3.3.1 Basic components of the Framework

The virtual market place is an environment that has a list of buyers and sellers. This list serves two purposes; i) helps the buyers to find the relevant sellers and ii) gives the number of buyers and sellers in the market for same product (supply and demand ratio). Buyers and sellers in the environment have access to following *knowledge* sources:

i) An Ontology (Domain Specific Knowledge)

ii) Set of general commonsense rules

iii) Case Base, where previous experiences of these agents are stored

Agents require world knowledge for product information[5], which is obtained from the ontology. Commonsense rule base in this framework drive the agents to negotiate sensibly, by guiding the agents to follow the basics buying behavior. Agents in this framework learn from experience, in the sense of Case based reasoning (CBR)

[5] Ideally these agents should consult recommender agents [36] for information about relevant counterpart (selling agent) for negotiation.

techniques[6]. Negotiations along with product information, market conditions and the outcome of negotiation might potentially be added to the case base for future negotiations. A high level view of the framework is shown in figure 3.1.

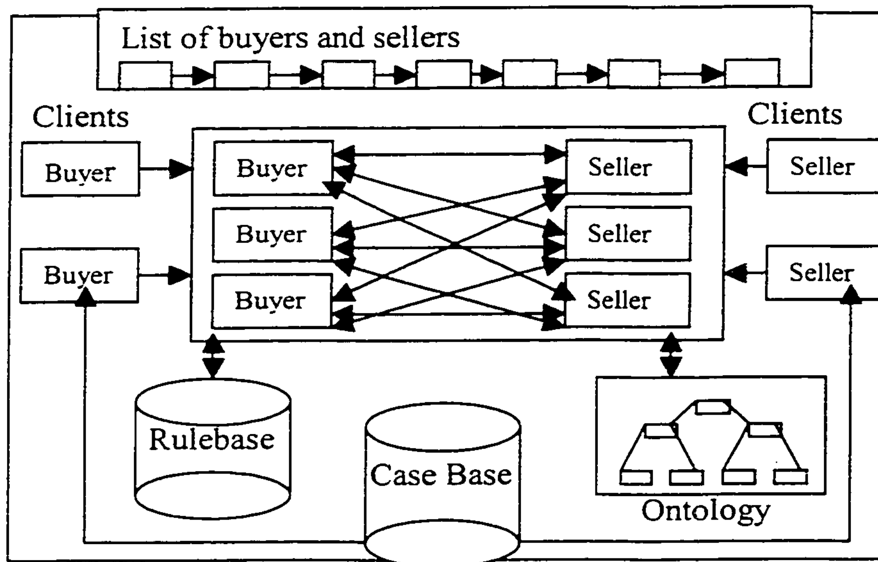


Figure 3.1: Basic components of the virtual marketplace

3.3.2 Overview of the negotiation process

The process starts when users (clients) create buyer and seller agents that are sent to the virtual marketplace. Buyers and sellers are registered in the marketplace where a list of buyers and sellers is maintained. In the current framework it is buyers that are proactive[7]; it is buyers that look for sellers and initiate a negotiation with sellers. An overview of the process from a buyer's perspective is described below.

- A buyer **b** is created by some user
- The buyer **b** enters the marketplace and registers in the environment

---

[6] See appendix for details on CBR techniques

[7] Modelling the sellers also proactive would lead to interesting scenarios, where these sellers look for relevant buyers and inviting them for negotiation; a way of modelling seller marketing strategies. We are not considering this issue in this work due to complexity issues, which are beyond the purpose of this research work.

- The buyer **b** retrieves a publicly available price range from ontology for the product in question
- Then buyer **b** retrieves its relevant experience from the case base
- Based on its attitude, publicly available price range, and experience buyer **b** computes its own price range as a complex function that we discuss later, in this chapter
- If the buyer mentioned price is less than the calculated agent max, then agent max is buyer mentioned price.
- Buyer **b** queries the environment (market) for a list of sellers, S, selling the same product
- For each relevant seller in the market, buyer **b** creates buyer clone **bc**
- Buyer clone **bc** sends an (asynchronous) message to each relevant seller $s \in S$ requesting a negotiation
- Seller provides a handle for a negotiation by creating seller clone **sc** for the respective buyer clone **bc**, or decline to negotiate
- Negotiation starts between each pair of **sc** and **bc**
- Buyers start bidding with their price range's minimum, while sellers start with their maximum (agent's price ranges are hidden)
- A deal is made when the buyer's maximum reaches the seller's minimum
- No deal is made if the buyer's maximum falls short of seller's minimum
- Both buyers and sellers might save the experience for future use

3.4 Definitions used in this Framework

3.4.1 Agents Attitude

An agent's attitude is a hidden mental state comprised of tripe (IOTime, IOPrice, Commit) representing the importance of time, the importance of price, and the commitment level of an agent. We believe that these three attributes play an important

role in purchasing decisions. Urgency and commitment reflects the desperateness for the purchase. Importance of Price determines the maximum user can spend for the purchase. It is assumed that IOTime, IOPrice, and Commit take on values in the open interval [0,1]. For example,

An agent with attitude = (1.0, 0.2, 0.8) represents,

i) Time is a priority

ii) Price is not important

iii) Commitment level is rather high

This attitude of a buyer is ideal from a seller's perspective. However, in this framework these attributes of an agent are hidden from other agents.

3.4.2 Public Price Range (PPR)

All agents in the marketplace are assumed to have access to a product price range that can be obtained from product ontology. Ontology[8] returns a public price range of min and max for the given product.

PPR (prod) = [pmin, pmax]

3.4.3 Supply/Demand Ratio

Supply and Demand Ratio represents the number of buyers and sellers present in the market during negotiation for certain product.

Supply/Demand Ratio = |S|/|B|

[8] In our model Ontology is a hierarchy of concepts. It supplies domain specific knowledge for the agents.

3.4.4 Negotiation

Negotiation is a process of offers and counter offers that can be in any of the following states,

(i) $DONE^+$: negotiation completed Successfully

(ii) $DONE^-$: negotiation completed Unsuccessfully

(iii) $DONE^0$: negotiation is still in progress

A negotiation record ({(Offer, Counteroffer)}) is an ordered list that contains,

(i) Offers and Counteroffers

(ii) An Outcome {$DONE^+$ or $DONE^-$}

3.4.5 Agent Experience

A new agent experience results after every negotiation. In addition to the negotiation record, an experience record contains information about the product information, the agent's attitude, the public price range, the agent's price range, and the market conditions (demand/supply ratio). The following is an example of a buying agent experience.

Product Category ⇒ Electronic Entertainment

Product Name ⇒ 36" TV

Warranty ⇒ 2years

Public Price Range ⇒ [1200, 2000]

Agents Price Range ⇒ [1250, 2000]

Attitude ⇒ [1.0, 0.5, 0.8]

SupplyDemandRatio⇒ (3)

Negotiation ⇒ ({(1250, 1800), (1400, 1600), (1500,1500)},DONE+)

This experience represents an agent's experience in buying a 36" TV, when the supply-to-demand was 3 to 1, the agent was highly committed to buying, the price was not much of a factor, but time was crucial. Under those circumstances, the negotiation was successfully completed after three exchanges of offers and counter offers. This experience could be used in the future when buying similar products.

3.4.6 Agent's Price Range (APR)

An agent's price range in this framework is a function of its attitude (refer to 3.4.1), public price range (refer to 3.4.2), supply demand ratio (refer to 3.4.3) and Warranty.

There are several functions that are plausible here. This price range could also be a function of its attitude, public price range, supply-to-demand ratio, and its experience. However note that the lower the minimum price range for a buyer the longer the negotiation will take place. Thus, the buyer's minimum price range must be higher than the public's minimum if the importance of time is high for that buyer. Moreover, when the commitment of a buyer is high the agent should be as close to the public's maximum as possible, since a high commitment indicates desperateness for the product. However, the importance of price should balance the degree of commitment (in a sense, a high commitment is a form of desperation, and the importance of price provides a sobering effect).

SupplyDemandRatio has slight affect on the buying and selling decisions of the agents. If the SupplyDemandRatio is low then the demand for the product is more. Therefore

buying agents should increase their price slightly. This reasoning is reflected in the suggested functions.

Warranty of the product also affects the price of the product. If Warranty is more in current situation than that of experience then buying agents should increase their max price.

Intuitively, a seller reasons in the opposite direction. The following two functions are one suggestion to capture the above reasoning, where [t (importance of Time), p (importance of Price), c (commitment)] represents an agent's attitude:

APR^{buyer} (PPR(product), (t, p, c), SDR, War_{cur}, War_{exp})

$= [\langle pmin + (t)(pmax\text{-}pmin)/\psi \rangle, \langle pmax - (pmax)(p)(1\text{-}c)/\psi + pmax(1\text{-}SDR)\,\varepsilon +$

$pmax(War_{cur}\text{-}War_{exp})\,\gamma \rangle]$

where, SDR_{cur} -SupplyDemandRatio in current situation,

War_{cur} -Warranty in current situation,

War_{exp} -Warranty in experience, and

ψ, ε, and γ are variables which vary based on the domain.

Note that PPR(refer to 3.3.3.2) returns *pmin* and *pmax* for a given product from the ontology.

APR^{seller} (PPR(product), (t, p, c), SDR, War_{cur}, War_{exp})

$= [\langle pmin + ((pmin)(p)(1\text{-}c)/\psi) + pmin(1\text{-}SDR)\,\varepsilon + pmin(War_{cur}\text{-}War_{exp})\,\gamma \rangle,$

$\langle pmax\text{-}(t)(pmax\text{-}pmin)/\psi \rangle]$

The following example explains the rationale behind functions: Let the product be a 36"TV, and the buying agents attitude be [1.0, 0.5, 0.8], SDR 0.5, Warcur is 1.5,

Warexp is 2. Assume that ψ is 10, ε is 0.01, and γ is 0.01 for this domain. Assume that PPR(36"TV) would return, *pmin* as $1200, *pmax* as $2000 from the ontology.

Then Agent Price Range (APR) for a buyer is,

$= [1200 + (1.0)(2000\text{-}1200)/10,\ 2000 - 2000\ (0.5)\ (1\text{-}0.8)\ /10 + 2000(0.5)(0.01) + 2000(1.5\text{-}2)(0.01)]$

$= [1200 + 80,\ 2000 - 20 + 10 - 10]$

$=[1280,\ 1980]$

This agent price range would be between 1280 and 1980. The agent would start its initial proposal at $1280 and his bidding price will go up to a maximum of $1980. The reason the buyer would start with a higher bidding price than the public minimum is that time is important for that buyer. Higher bids would reduce the time a negotiation takes.

In contrast, for a seller it works in the opposite direction. For example, consider a situation where the seller has the same attitude as the buyer in the above case and it is trying to sell the same product as above.

Then the Agent Price Range (APR) for the seller is,

$= [1200 + (1200)\ (0.5)\ (1\text{-}0.8)\ /10 + (1200(0.5)(0.01)) + (1200(1.5\text{-}2)(0.01)),\ 2000 - (1.0)\ (2000 - 1200)/10]$

$= [1200 + 12 + 6 - 6,\ 2000 - 80]$

$= [1206,\ 1920]$

This selling agent would have a price range between 1206 and 1920. Since the time is important for the seller, he would start with a counterproposal $1920. The higher the price decrements, the faster the product is sold.

3.5 The negotiation process in detail

A Buyer **b** with an attitude (bt, bp, bc) and a Seller **s** with an attitude (st, sp, sc) enter the marketplace. Consequently,

- **b** computes its price range: $[bpr_{max}, bpr_{min}] \leftarrow APR^{buyer}$ ([pmin,pmax], (bt, bp, bc)
- **s** computes its price range: $[spr_{max}, spr_{min}] \leftarrow APR^{seller}$ ([pmin,pmax], (st, sp, sc)
- **b** hides its bpr_{max} and starts its bidding with $\mathbf{b}_{bid} \leftarrow bpr_{min}$
- s hides its spr_{min} and starts its bidding with $\mathbf{s}_{bid} \leftarrow spr_{max}$
- With each successive offer/counteroffer buyers and sellers update their respective biddings as follows: $\mathbf{b}_{bid} \leftarrow (\mathbf{b}_{bid} + \alpha)$ and $\mathbf{s}_{bid} \leftarrow (\mathbf{s}_{bid} - \beta)$
- α and β are the buyer's step increment and the seller's step decrement, respectively. α and β are calculated using their experience and present market conditions. A good α and β for buyer and seller respectively, will lead to more success and profitable negotiations.
- A negotiation is always in one of the following states:

  (i) $DONE^{+}$ if $\mathbf{b}_{bid} \geq \mathbf{s}_{bid}$;

  (ii) $DONE^{-}$ if $bpr_{max} < spr_{min}$ (these two are hidden form each other)

  (iii) $DONE^{0}$ if $(\mathbf{b}_{bid} < \mathbf{s}_{bid}) \wedge (bpr_{max} \geq spr_{min})$

In accordance with the process discussed in 3.3.2, it must also be noted that in general a buyer has more than one negotiation thread running concurrently (one thread with every potential seller). Similarly, a seller also has more than one negotiation thread. The buyer waits for each negotiation thread to return a result $r \in \{(DONE^+, price), (DONE^-, price)\}$. If the time is more important than price, the buyer exits the marketplace (terminating all its clones) as soon as a result $r = (DONE^+, price)$ is received, otherwise, the buyer waits for all negotiation threads to terminate and selects the one that found the best deal (if any)[9]. This centralized control of parent agents over the clones helps the agents to be consistent in finalizing the deals in a distributed environment. However, currently this framework is built on a single sequential machine.

3.6 Case-Based Reasoning and Learning from Experience

Case-Based Reasoning (CBR) is a tool used in this framework, where negotiating experiences are captured and reused in similar future negotiations. This experienced-based approach provides adaptive negotiation strategies that can be generated dynamically. It is well known that experience is an integral part of the learning process. It has been suggested that most human learning is from experience [14]. Experience makes humans good negotiators.

The agents in this model learn from their experience over a period of time and they reason accordingly during negotiations. Regardless of the outcome of the negotiation, both buyers and sellers occasionally save their experience in a case base for future use.

[9] At this moment there is no bilateral communication between the buyer's threads (clones). Such an extension adds considerable complexity to the model, although it does open up interesting possibilities to explore.

Learning from failure is considered in this framework, as unsuccessful negotiations are also stored for future use.

We clearly state that the issue of learning in this framework is referred, in the sense of learning in Case Based Reasoning. Within artificial intelligence (AI), learning usually means the learning of generalizations, either through inductive or through explanation-based means. According to [14], Case-Based Reasoning achieves most of its learning,

1. Through the Accumulation of new cases
2. Through the assignment of indexes

New cases give the reasoner additional familiar contexts for solving problems or evaluating situations. This additional knowledge they gain in due course of time is referred as "learning". New indexes allow a reasoner to fine-tune its recall apparatus so that it remembers cases at more appropriate times.

This framework has very limited emphasis on indexing of cases, as the case in this framework is simple. Accumulation of cases and updation of cases contribute to "learning" in this framework. Currently, agents in this framework use their previous experience to adjust their attitude. But many other strategies could be incorporated in this framework, so that the agents could reason from different factors in their previous experience. When dealing with a case base one has to carefully consider a strategy for the following steps in case-based reasoning [14]:

(i) Case representation

(ii) Case indexing and retrieval

(iii) Case adaptation

(iv) Case evaluation and updation

A case (experience) in our model has the following structure

⟨ProductCategory: prod, (e.g., Electronic Entertainment)

ProductName: pname, (e.g., 36" TV)

Warranty: war, (e.g., 2 years)

PublicPriceRange: ppr, [pmin, pmax]

AgentPriceRange: apr, [amin, amax]

Attitude: att, (e.g., ⟨1.0, 0.2, 0.8⟩)

SupplyDemandRatio: sdr,(e.g., ⟨3⟩)

Negotiation: neg (⟨{⟨1250, 1800⟩,⟨1400, 1600⟩,⟨1500,1500⟩}, DONE+⟩)

⟩

3.6.1 Matching Cases

Cases are indexed in the case base by the product name. When searching for 'relevant' cases (or experiences), a perfect match cannot be expected. List of cases are retrieved from the case base, for each similar product retrieved from the ontology. Similar cases (or experiences) retrieved in to the list contain both successful and failed negotiations. Out of all the retrieved cases from the list, the case that matches the current scenario to the most is the 'relevant' case (or experience). When searching for relevant experience, cases are matched as follows.

$$Match(c_1, c_2) = 1/3\,(PS(prod(c_1), prod(c_2)) + AS(att(c_1), att(c_2)) + RS(ppr(c_1), ppr(c_2)))$$

Where *PS : Product Similarity,*
AS: Attitude Similarity,
RS: Public Price Range Similarity, and
c_1, c_2 *are present scenario and similar experience respectively.*

In calculating match value, this framework uses similarity criteria on different attributes. The range of similarity is an interval of [0.1...1.0]. The attributes to be matched include product, attitude of the agent, and the public price range. This framework does not consider the SupplyDemandRatio in matching a relevant experience, but it does consider SupplyDemandRatio in calculating agent price ranges. First, a brief discussion on the similarities mentioned in the definition of match function.

3.6.1.1 Product Similarity (PS)

Matching the similarity of two products is very important in matching an experience, as humans use their previous experience in buying or selling 'related' products. Here related product does not mean that it is just the 'conceptual similarity' between the two products. The reason for this can be explained with the following example:

"When buying a scanner, one might recall their experience in buying a printer. In this case the conceptual similarity between the product categories seems to be sufficient. However, this is a very simplistic view, since one would hardly recall their experience in buying a (computer) mouse when one is buying a (computer) monitor, although both are "computer products", conceptually. Clearly, the price range is also crucial. That is, our experience in buying big items with similar price ranges might be similar even though the product categories might be different. The similarity between two products might therefore be a function of both conceptual product similarity and its price.

ProductSimilarity PS (pr1, pr2, prod1, prod2) = (PrS (pr1, pr2)) * (CS(prod1, prod2))

where PrS and CS are defined as follows:

3.6.1.1.1 Price similarity (PrS)

The following function is one suggestion as how to compare the similarity between two product prices.

$$PriceSimilarity(pr1, pr2) = \left(1 - \frac{abs(pr_1 - pr_2)}{\max(pr_1, pr_2)}\right)$$

where pr_1 and pr_2 are the average price of Public Price Range for product1 and product2 respectively.

For example, a product1 price is $100 and another product2 price is $120 then the similarity between the prices of these two products is,

$$PriceSimilarity(pr1, pr2) = \left(1 - \frac{abs(100 - 120)}{120}\right)$$

$$= \left(1 - \frac{20}{120}\right)$$

$$= 0.83$$

Therefore the price of these products is very similar.

3.6.1.1.2 Conceptual Similarity (CS)

The conceptual similarity between two products is measured from the ontology[10]. Agents in the virtual marketplace have the access to a domain-specific ontology of product information. Using the notion of semantic distance in a semantic network, the simple measure of conceptual similarity between two products may be measured as follows:

---

[10] Note that public price range (3.4.2) is also obtained from the ontology (domain specific knowledge) in this model. Currently, the ontology and the domain knowledge are quite limited. The ontology must be considerably extended to support recommender agents [36].

ConceptualSimilarity CS (prod1, prod2)

= 1/(*dist(prod1*, lub(*prod1, prod2*)) * 0.5 + *dist(prod2*, lub(*prod1,prod2*))*0.5)

where 'lub' is the least upper bound of two concepts in the ontology, and the distance between two concepts, dist(c_1, c_2), is the number of '`isa`' links from c_1 to c_2.

For example, assume that a monitor and printer are Computer products and the least upper bound between these two concepts is 1. Then the distance from printer to Computer products is 1 and the distance from monitor and Computer products is also 1. Conceptual similarity between these products is computed as follows:

ConceptualSimilarity CS (monitor,printer)

= 1/(dist(printer, lub(printer,monitor)) * 0.5 + dist(monitor, lub(printer,monitor)) * 0.5)

= 1/(dist(printer, Computer Products)) * 0.5 + (dist(monitor, Computer Products)) * 0.5)

=1/(1 * 0.5) + (1 * 0.5) = 1 (Printer and Monitor are Very Similar conceptually)

However, the conceptual similarity values could be adjusted according to the domain in modeling the ontology. For a detailed discussion on Conceptual Similarity (CS) refer to section 4.3.2 in chapter 4.

Since Prs and CS are explained in detail, consider an example to match product similarity of two products monitor and printer with price 220 and 260 respectively.

ProductSimilarity PS (pr1, pr2, prod1, prod2) = (PrS (pr1, pr2)) * (CS(prod1, prod2))

$$PS\ (220, 260, monitor, printer) = (PrS(220, 260)) * (CS(monitor, printer))$$

As explained above, PrS would return 0.91, as the price is close and CS would return 1, as monitor and printer are conceptually very similar products. Therefore PS would return,

$$PS\ (220, 260, monitor, printer) = 0.84 * 1$$
$$= 0.84$$

These two products are very similar as PS returns a value of 0.84, which is good threshold for matching two similar products. However, this match threshold may vary from domain to domain.

3.6.1.2 Attitude Similarity (AS)

Matching two attitudes has considerable significance, as attitude play major role in human buying behavior. The similarity between the attitudes of two agents is computed as follows:

$$AS\ (\langle t_1, p_1, c_1\rangle,\langle t_2, p_2, c_2\rangle) = ((1 - | t_1 - t_2|) + (1 - | p_1 - p_2|) + (1 - | c_1 - c_2|))/3$$

where $\langle t_1, p_1, c_1\rangle,\langle t_2, p_2, c_2\rangle$ are attitudes of agent1 and agent2 respectively.

For example, the attitudes of two different agents,

A1 is ⟨ 0.5,	1.0,	0.1 ⟩
Time is not really important	Price is important	Not committed

A2 is ⟨ 1.0,	0.1,	1.0 ⟩
Time is very important	Price is not important	Highly committed

The similarity between these two attitudes is computed as

$= ((1 - |0.5 - 1.0|) + (1 - |1.0 - 0.1) + (1 - |0.1 - 1.0))/3$

$= ((1 - 0.5) + (1 - 0.9) + (1 - 0.9))/3$

$= (0.5 + 0.1 + 0.1) / 3$

= 0.233

Therefore these two attitudes of A1 and A2 are not similar at all. Conversely, consider another example where attitudes of A1 and A2 are ⟨1.0,0.1, 1.0⟩ and ⟨0.9,0.2, 0.8⟩. Then the similarity of these two attitudes is,

= ((1 - |1.0 – 0.9 |) + (1 - |0.1 – 0.2) + (1 - |1.0 – 0.8))/3

= ((1- 0.1) + (1- 0.1) + (1 – 0.2))/3

=(0.9 + 0.9 + 0.8)/3

= 0.866

These two attitudes are very similar, as the match value meets the threshold for matching two similar attitudes.

3.6.1.3 Public Price Range Similarity (RS)

Computing RangeSimilarity is important in matching cases as price range is a very important attribute in purchasing decisions. The similarity between two public price ranges is computed as follows:

$$RS([min1, max1], [min2,max2]) = \left(1-\frac{abs(min_1 - min_2)}{max(min_1, min_2)}\right)\left(1-\frac{abs(max_1 - max_2}{max(max_1 + max_2)}\right)$$

For example, the given range of two different products, product1 and product2 is [1000,1500] and [2000, 2500] respectively. The similarity between these two ranges is computed as

$$RS([1000,1500],[2000, 2500]) = \left(1-\frac{|1000-2000|}{|2000|}\right)\left(1-\frac{|1500-2500|}{|2500|}\right)$$

$$= \left(1-\frac{1000}{2000}\right)\left(1-\frac{1000}{2500}\right)$$

$$= (1-0.5)(1-0.4)$$

$$= 0.3$$

These two ranges are not similar as the value returned by RS is 0.3, which fall short of the threshold for range similarity.

3.6.1.4 Matching most 'relevant' case

The most relevant case to the current scenario is retrieved by from the given list of cases using the following function, as mentioned in section 3.6.

$$Match\ (c_1,c_2) = 1/3\ (PS(prod(c_1), prod(c_2)) + AS(att(c_1), att(c_2)) + RS(ppr(c_1), ppr(c_2)))$$

For example, the attributes in the $\mathbf{c_1}$, current scenario (problem) include,

Product : Monitor

Attitude: $\langle 1.0, 0.1, 1.0 \rangle$ (seems a sense of urgency for the product)

Public Price Range: [200, 350]

Price : 275 (note that its an average of pmin and pmax in this model)

The attributes of $\mathbf{c_2}$, previous case (experience) in the Casebase include,

Product : Printer

Attitude: $\langle 0.8, 0.3, 0.8 \rangle$

Public Price Range: [200, 400]

Price : 300

The match between these two cases is computed as an average of the values returned by the functions PS, AS, and RS. In this example, PS returns 0.86, AS returns 0.8, and RS returns 0.96, using the methods discussed in previous subsections. Now the match returns the value of the match as follows:

$$Match\ (c_1,c_2) = 1/3\ (0.86 + 0.8 + 0.96)$$

$$= 0.87$$

This previous experience could be used in solving the current problem as the match between c_1 and c_2 is 0.87, which is considered as a good threshold for a relevant experience.

3.6.2 Learning from experience

Agents in this framework use their previous experiences to adjust their attitudes slightly and calculate their price ranges, to calculate bid increments (for buyers) and bid decrements (for sellers)) in offers and counteroffers. Using a similar previous experience, attitude of the agent is slightly adjusted. This new adjusted attitude helps the agent to act more judiciously in calculating its range and bid increments. The reasoning behind this process is to bias these agents towards successful negotiations with in their constraints (not deviating from the current attitude and needs). This bias may help the agents to come up with a price range and α' (bidding increments and decrements) that maximizes probability of number of successful negotiations.

However, one must consider other learning strategies that could be incorporated in this framework to make these agents better negotiators.

3.6.2.1 Adjusting attitude

The attitude of an agent is adjusted based on previous experience. The experience we use here is the best case that matched with best threshold by the match function in section 3.6.1. The idea of using more than one case needs serous consideration in adjusting the attitude. It would be ideal to look at both positive and negative experiences at the same time and then make a decision in adjusting its attitude.

Finding optimal attitude is an important part of the learning process, as it leads to more successful negotiations. In calculating optimal attitude for a given situation, the agents

should not deviate completely from their clients needs. Therefore the adjusted attitude is always close to its initial attitude within certain threshold. As mentioned earlier, these agents learn from both successful and unsuccessful experiences. Following function is a suggestion to adjust the attitude of a buyer agent using both successful and unsuccessful experiences in a negotiation:

In case of using a successful experience, Importance of Time in current situation (t_{cur}) could be adjusted as,

$$t'_{cur} = t_{exp} \quad \text{if outcome(neg)} = \text{DONE}^{+} \text{ and } abs(t_{exp} - t_{cur}) \leq 0.1$$

$$t'_{cur} = t_{cur} + 0.1 \quad \text{if outcome(neg)} = \text{DONE}^{+} \text{ and } t_{exp} > t_{cur}$$

$$t'_{cur} = t_{cur} - 0.1 \quad \text{if outcome(neg)} = \text{DONE}^{+} \text{ and } t_{exp} < t_{cur}$$

Similarly, the Importance of Price in current situation (p_{cur}) could be adjusted as,

$$p'_{cur} = p_{exp} \quad \text{if outcome(neg)} = \text{DONE}^{+} \text{ and } abs(p_{exp} - p_{cur}) \leq 0.1$$

$$p'_{cur} = p_{cur} - 0.1 \quad \text{if outcome(neg)} = \text{DONE}^{+} \text{ and } p_{exp} < p_{cur}$$

$$p'_{cur} = p_{cur} + 0.1 \quad \text{if outcome(neg)} = \text{DONE}^{+} \text{ and } p_{exp} > p_{cur}$$

Similarly commitment could also be adjusted based on previous experience. It is interesting to note that in adjusting both t_c and t_p in above functions is very similar, the relation between t_c and p_c is inversely proportional in this model. Reasoning behind this phenomenon is that the agents tend to increase their probability of success using the successful experience as a guideline. For example, if importance of time is high in experience when compared to importance of time in current situation, then agents tend

to increase their Importance of time, as they are aware from the experience that they could achieve success by slightly increasing their urgency[11].

Similarly, if the Importance of price is more in experience when compared to importance of price in current situation, then agents tend to increase their Importance of Price, as they are aware from the experience that they could achieve success with less price. As it is observed in the functions that the difference between adjusted attitude and initial attitude is bounded by a threshold of 0.1. We believe that this threshold is a reasonable measure in controlling the agents, not to deviate from the actual needs of their clients.

In case of using a negative experience the Importance of Time in current situation could be adjusted as,

$$t'_{cur} = t_{cur} + 0.1 \qquad \text{if outcome(neg)} = \text{DONE}^- \text{ and } t_{exp} > t_{cur}$$

Similarly, in case of Importance of Price in current situation could be adjusted as ,

$$p'_{cur} = p_{cur} - 0.1 \qquad \text{if outcome(neg)} = \text{DONE}^- \text{ and } p_{exp} < p_{cur}$$

In contrast to what we observed in using successful experience, the agents tend to behave completely different in using negative experiences (unsuccessful experiences). In using a negative experience, if the Importance of Time in experience is higher than Importance of Time in current situation, then agents have to increase their importance of time slightly higher as they are aware from experience that their Importance of Time is not enough to achieve success. In contrast, if the Importance of time is less in

[11] Note that the Importance of Time and Importance of Price affect the price ranges of agents and their increments as discussed earlier.

experience when compared to Importance of time in current situation, they would stick with their own Importance of time as it is already better than that of experience.

Similarly, in case of Importance of Price is lesser in experience than that of current situation, then the agents should reduce their Importance of price, as they are aware from experience that their Importance of Price is too high to make a successful deal in the market.

Thus the agents adjust their attitude to find an optimal attitude using their experience, but not compromising on the needs of the user and then calculate their price ranges that reflect the experience, and their adjusted attitude. Adjusting attitude could be a much more complex functions than what suggested above. In the above function, we only considered the attitude of the agent in its experience in adjusting current attitude. However, the success or failure of a negotiation is determined by many factors, not just agent's attitude. The attitude adjustment function needs more investigation to identify the possible other factors which determine the result of the negotiation.

3.6.2.2 Calculating Agent's Price Range

After adjusting their attitude the agents calculate their maximum and minimum price range using the function Agent's Price Range (APR), described in section 3.4.6.

3.6.2.3 Calculating Bid Increment α

Calculating bid increment α is a very important issue as it determines result of the negotiation as well as length of negotiation. Higher bid increments would reduce the length of negotiation (offers and counter offers). Finding an optimal α, would increase the success rates of agents in negotiation. Following function is a suggestion to calculate bid increment.

$$\text{Bid Increment (Att', APR)} = \left\langle \frac{Apr_{max} - Apr_{min}}{5} + 0.5 * \left(\frac{t'(Apr_{max} - Apr_{min})}{10 * p'} \right) \right\rangle$$

where p' = 0.1, if p' = 0.0 and APR = $[Apr_{min}, Apr_{max}]$

This function assumes that an average bid increment, 20 % of difference of the price range. On top of that, factors like Importance of time and Importance of Price would affect the bid increment. The above function is a reflection of the fact that Importance of Time is directly proportional to bid increment and Importance of Price is inversely proportional to bid increment. However, we do admit that most of these numbers used in these functions are ad-hoc, and in fact these numbers might have to be product dependent.

After calculating bid increment, the value is added to the AprMin and sent as proposal to the opponent in case of a buyer. Intuitively, it works opposite for a seller. They continue negotiation until buyer's proposal matches or greater than seller's ArpMin.

3.6.3 Saving experience for future use

Agents save their experience from time to time for future use. When a negotiation terminates, a search is done in the case base and the new experience is matched with the ones in the case base. When searching for updating or adding a new experience a match between two cases is done as follows:

$$Match\ (c_1, c_2) = \frac{1}{4}\begin{pmatrix} w_1 \times \mathrm{PS}(\mathrm{prod}(c_1), \mathrm{prod}(c_2)) + w_2 \times \mathrm{AS}(\mathrm{att}(c_1), \mathrm{att}(c_2)) + \\ w_3 \times \mathrm{RS}(\mathrm{ppr}(c_1), \mathrm{ppr}(c_2)) + w_4 \times \mathrm{NS}(\mathrm{neg}(c_1), \mathrm{neg}(c_2)) \end{pmatrix}$$

where NS is *Negotiation Similarity*, AS is *Attitude Similarity*, RS is *Price range Similarity*, and PS is *Product Similarity*.

3.6.3.1 Negotiation Similarity

Negotiation similarity is measured in terms of the number of offer exchanges in the negotiation and the negotiation result. Number of exchanges reflects the time taken for a negotiation and the result of negotiation is a major attribute in comparing two different negotiations. It is important to give equal priority to the result of negotiation and number of exchanges taken place during the negotiation. The similarity between two negotiations is defined as follows:

$$NS\ (\langle L_1, outcome_1\rangle, \langle L_2, outcome_2\rangle)$$

$$= \begin{cases} 1 - ((abs(N1 - N2)/\max(N1, N2)) \times 0.5 + 0.5) & \text{if } outcome_1 <> outcome_2 \\ 1 - abs(N1 - N2)/\max(N1, N2) & \text{otherwise} \end{cases}$$

where L1 and L2 are lists of offers and counter offers, N1 and N2 are number of exchanges in Negotiation1 and Negotiation2 respectively. The above function assigns a weight of 0.5 each to outcome and number of offer exchanges, there by it adds 0.5 if the outcome is not the same.

Though it sounds exciting to use this approach, assigning weights to attributes in the above *Match* function is a major challenge, which needs thorough investigation to model and reflect the human memory refining process in buying and selling process. Currently, this model assumes equal weight for all attributes. Further investigation to test various weighting schemes, perhaps using a machine learning experiments could help in finding optimum weight assignments to the given attributes.

3.6.3.2 Updating Casebase

Efficiency of this system depends on storage of cases in the Casebase. One has to be judicious in storing new cases, so that Casebase is not populated exponentially. Efficient way storing of cases result in conceptual gain as well as computational gain. Conceptual gain is that the agents refine their memory with by adding new things to "what they already know". Computational gain is that the system will be faster as retrieval of cases is reduced.

After matching the new experience with existing cases in the case base, the new case is stored as a novel experience, if a strong match is not found in the case base. When a strong match occurs, the two cases are merged resulting in a modification of an existing experience, *case retainment.* The following function is a suggestion to merge two cases:

$$Merge(\langle p1, pn1, ppr1, wr1, pr1, att1, sdr1, neg1\rangle, \langle p2, pn2, wr2, ppr2, pr2, att2, sdr2, neg2\rangle)$$

$$= \begin{bmatrix} \text{lub}(p1, p2), (pn1 = pn2), avg(wr1, wr2), (ppr1 = ppr2), avg(pr1, pr2), \\ avg(att1, att2), avg(sdr1, sdr2), \min(neg1, neg2) \end{bmatrix}$$

Merge takes the attributes of both old case and new case and merge them together to get a hybrid of both the cases. Merging of two cases occurs only when both products are same and the price range is similar. Then the average of price, attitude and SupplyDemandRatio of the two cases is computed. Then the negotiation that took the minimum time is selected. All these attributes will constitute a refined experience and old case is replaced by the refined experience. One needs to consider the result of the

negotiation in merging cases, as result of the negotiation affects bidding patterns in negotiation as discussed in the end of section 3.6.2 in this chapter. Merge is explained in detail with an example in Appendix D. However, this merge function is a suggestion how one could refine their experience. The validity of this suggestion needs to be tested thoroughly in comparison with various plausible approaches in refine memories.

3.7 Conclusion

In this chapter, we proposed a framework of a virtual marketplace where buying and selling agents with attitude, learn from their experience and negotiate autonomously on behalf of their clients. The initial attitude of the agents is three important attribute of mental state of the user, which includes importance of time vs. importance of price and commitment. Then these agents adjust their attitude and calculate price ranges using complex functions of prior experience, market conditions and product characteristics. Retrieval of experience is done using fuzzy measures and functions. Case Based Reasoning techniques are used to model learning from experience.

The agents in this framework are goal oriented and they compete and coordinate for their benefits. The interactions between agents in this framework are many to many. These agents tend to store their experience for future use. They refine their experience from time to time for efficient functionality. The framework proposed in this chapter is an attempt to model limited aspects of human buying behavior using attitude and experience as major factors.

Chapter 4

Framework Design and Implementation Issues

The model discussed in the previous chapter is an attempt to automate negotiation in e-commerce transactions. This chapter focuses on the plausibility of implementing the ideas discussed in last chapter.

4.1 Building the environment (virtual market place)

The environment in this model supports the creation and disposal of agents. Users create agents at their will send them in to the marketplace for negotiation. There are two lists in the environment for buying and selling to register in the environment after they enter the market (environment).

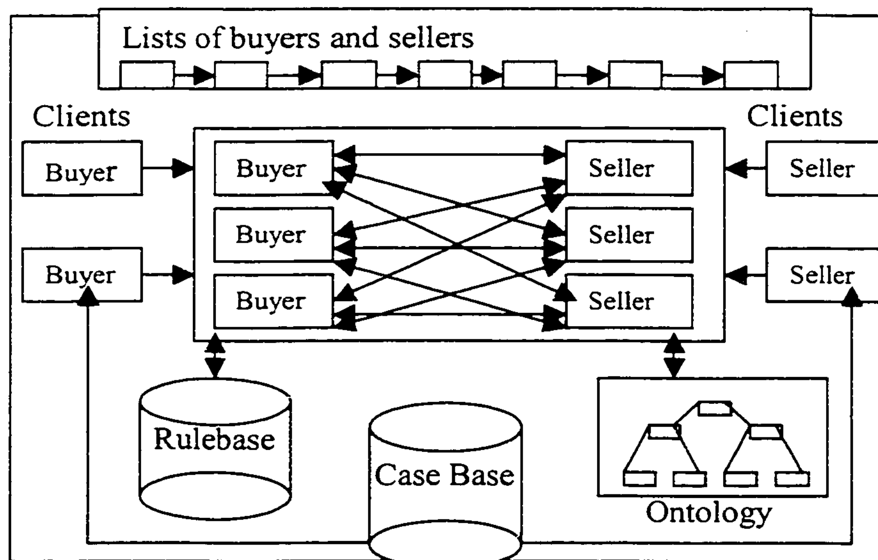


Figure 4.1: Basic components of the virtual marketplace (Environment)

4.2 Object Diagram of the model

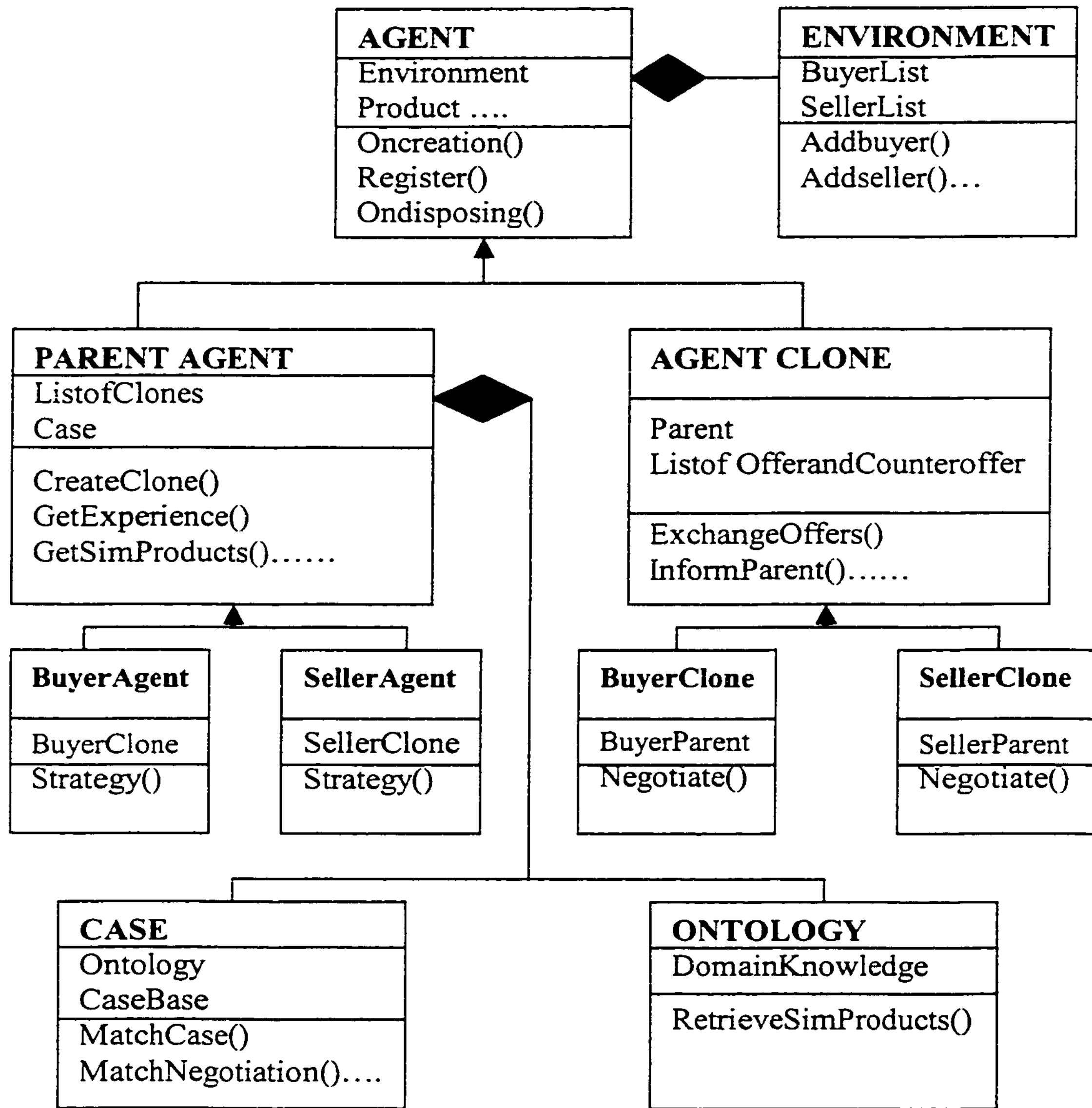


Figure 4.2: Object Diagram of the Virtual Marketplace

Major Objects in this model include, Agent, Parent agent that is an extension of Agent, Case, and Ontology. Buying and Selling agents created in this virtual marketplace are of type Parent agent. Agent clones are also an extension of Agent type. Buyer agent and Seller agent interact with environment to register and to get the list of relevant

negotiation parties. Interaction between Parents and Clones is mutual as there is exchange of messages during the process. Buyer agent and Seller agent interact with Case to retrieve relevant experiences and also interact with ontology to retrieve public price ranges. Object Case interacts with Ontology to retrieve list of similar products. The interaction between these objects is shown in the following figure:

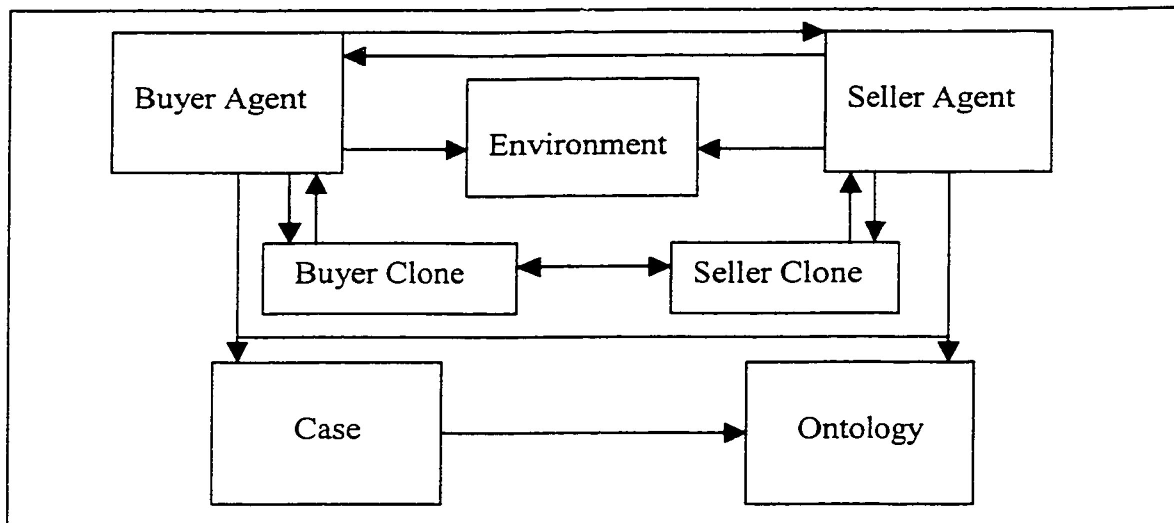


Figure 4.3 Object interactions in the proposed Framework

4.3 Implementing the process of negotiation

In a nutshell the process of negotiation is as follows: Users create buyers and sellers in the virtual market and these agents register themselves in the market. Buyer and Seller agents retrieve their relevant experience. Buyer agents look for relevant sellers in the market from the list. Then Buyer agents create clones based on number of relevant sellers in the marketplace. These clones interact with the Seller agent requesting to negotiate; in turn the Seller agent creates clones to negotiate, if the proposal is acceptable. Buyer and Seller clones negotiate to make a deal.

This process is divided in to four parts from an implementation point of view,

(1) Creation of agents and clones

(2) Retrieval of Similar products from ontology

(3) Retrieval of relevant experience

(4) Implementation of the functions discussed in previous chapter

4.3.1 Creation of Agents and Clones

Initially, we planned to create the agents in IBM's ASDK. IBM's ASDK supports most of the required features like Aglet proxies (for creation of agents), Cloning, Communication methods for agent interaction, code mobility, security, and distributed environment. After thorough study of the package, we realised that use of the package in our model would add more overhead in implementation.

After agents are created in the marketplace, Buyer agents create their clones matching the number of relevant sellers in the market. In turn, Seller agents create clones if the proposal from the Buyer Clone is acceptable. Then the Buyer and Seller clones negotiate to reach a deal. The purpose behind creating clones is to achieve computational speed, and also there is a conceptual gain that agents may find faster deals where time is priority.

Buyer agents request the Seller for negotiation and Sellers in turn respond to the request by creating a Seller clone for negotiation. Buyer Clones and Seller Clones negotiate in accordance with their Parent's strategy. Parents have a handle on their clones for communication. These clones return the result after the end of negotiation. Buying and Selling agents evaluate the results returned by clones and decide on 'who to make deal with' based on their strategy.

4.3.2 Retrieving Similar products form Ontology

Ontology serves two purposes in this framework as mentioned in previous chapter. Agents retrieve public price range from ontology. In matching cases ontology is used to match conceptual similarity of two products. Domain specific knowledge assumed in this framework is very limited. In case of large-scale implementations, one needs to supply vast amount of domain specific knowledge for efficient results. This framework uses a very basic ontology just for testing purposes. The ontology assumed in this model is represented in a database. There are two databases representing the ontology in this model. One is to retrieve the public price range and the other is to capture the conceptual similarity between two products. For example, a consumer electronics domain is represented conceptually as shown in the following figure.

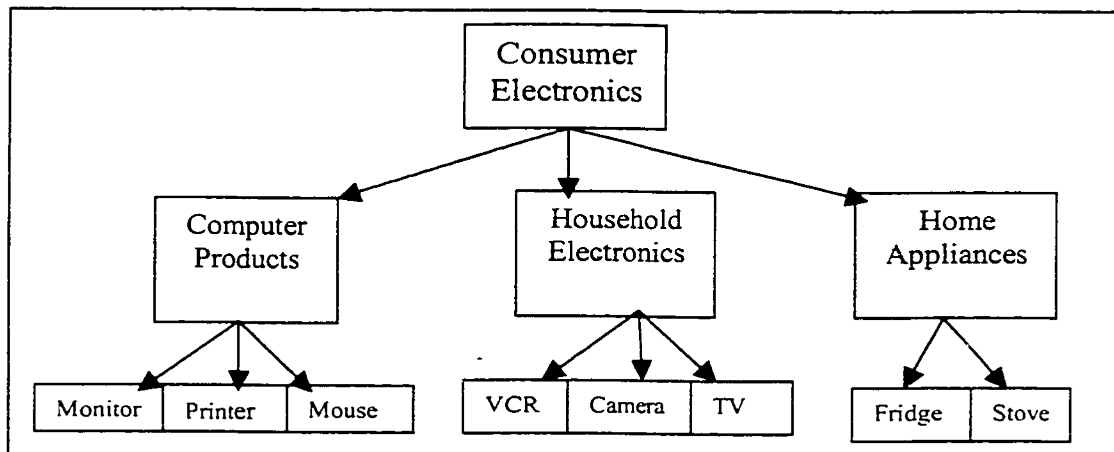


Figure 4.4: Sample Ontology for Consumer Electronics Domain

In the above ontology, the value of conceptual similarity of two products ranges between an interval $\langle 0.0, 1.0 \rangle$. Products like monitor, printer and mouse belong to the same category, Computer Products. Therefore the conceptual similarity between these two products is 1.0. In case of comparing a monitor and a TV the conceptual similarity

is 0.5, as monitor and TV are represented under two different concepts, Computer Products and Household Electronics respectively. Similar reasoning applies to Fridge and Stove when compared with monitor or TV.

This implementation is not considering the formal approaches for ontology modeling, as the focus of this model is learning and intelligence, not ontological issues. For testing purposes, the ontology is modeled in a relational database where the value of conceptual similarity between two products is represented directly, as the assumed ontology is very limited. The key to retrieve the value is combination of both product categories. Another database is used to represent the public price range for the products in question. The ontology assumed in figure 4.5 is represented in the form of database tables as shown in the following table:

Product Cate	Prod Name	Minpr	Max Pr
Comp Prods	Monitor	250	350
Comp Prods	Printer	300	450
Comp Prods	Mouse	8	15
Home Elecs	Vid Cam	500	700
Home Elecs	TV	300	500
Home Elecs	VCR	100	200
Home Appl	Fridge	400	550
Home Appl	Stove	300	450

Product Category1	Prod Category2	Value.
Comp Prods	Comp Prods	1
Comp Prods	Home Elecs	0.5
Comp Prods	Home Appl	0.5
Home Elecs	Comp Prods	0.5
Home Elecs	Home Appl	0.5
Home Elecs	Home Elecs	1
Home Appl	Home Appl	1
Home Appl	Home Elecs	0.5
Home Appl	Comp Prods	0.5

Table 4.1 Database tables representing the Conceptual hierarchy

The table on the left in table 4.1 represents the public price range. Min and Max price of a product is retrieved using Prod Name as key. Conceptual similarity is represented in the table on the right in table 4.1. Conceptual similarity value is retrieved using

combination of product category as key. This modeling is convenient, as the assumed ontology is very limited. In case of large-scale implementations, one should rely on formal techniques to represent ontologies, so that retrieval of information is efficient.

4.3.3 Retrieval and Adaptation of relevant experience

Buying and selling agents probe the case base on entry in to the marketplace to retrieve relevant experience. This framework uses Case Based Reasoning techniques to inculcate learning from experience in these agents. Case represents specific knowledge tied to specific situation, and represent knowledge at an operational level [14]. It is necessary to store all crucial facts about each negotiation episode in the form of a case, so that this knowledge helps agents in generating better proposals and achieving successive deals. Representing and indexing cases, adapting old experience in new situation, evaluating the new solution, and finally updating the Casebase are important steps in Case-based Reasoning.

4.3.3.1 Representing Cases (Experience)

Cases in this model are represented in a relational database as shown in the following table:

Pcat	Pname	War	PPR	Price	Attitude	SDR	NegRec	Res
Ele Ent	36"TV	2	1200-2000	1500	1.0,0.2,0.8	2	1250/1800/1400/1600/1500/1500	1
Ele Ent	28"TB	2	1000-1500	1200	0.9,0.2,0.9	1	1040/1480/1120/1410/1200/1300	0
Com pro	Monitor	1	200-350	260	0.5,0.8,0.8	0.5	200/310/220/290/260/260	1

Table 4.2: Database table representing Cases (experiences)

In the above table first 6 attributes are straightforward. Attribute NegRec represents the offers and counteroffers as one string, which is parsed in to individual attributes on

retrieval. Attribute Res represents the result of negotiation and has a value 1 or 0 representing success or failure respectively.

4.3.3.2 Choosing Relevant Experience

Matching similar cases after retrieval is interesting in this model, as matching attributes of an experience and current situation is a measure of a function and then matching the whole case is a measure of the average value returned by these individual functions, explained in section 3.6.1 of chapter 3. If the average value returned functions, matches the necessary threshold, then that case is used in generating the new solution.

4.3.3.3 Adapting a Case (an Experience)

As discussed in section 3.6.2 of chapter 3, calculation of 'α', which is used in making offers and counter offers is calculated from negotiation record Neg Rec. The string retrieved from NegRec need to be parsed in to individual attributes to calculate average bidding price 'α'.

The reason for storing the whole negotiation is that it is possible to predict the behavior of the opponent using the knowledge from each offer and counteroffer in experience. This idea is not addressed in this model, but being explored by other researchers in [15,6]. These ideas may be incorporated in future work in this model

4.3.3.4 Updating Casebase (Refining Experience)

It is necessary to update the Casebase after each negotiation episode, so that the experience could be used in future endeavors of these agents. As discussed in section 3.6.3 of chapter 3, the new solution is evaluated and matched with the existing cases in Casebase. If the new solution matches to a certain threshold to the existing cases in

Casebase, then case in the case base is updated using the function suggested in section 3.6.3.2 of chapter 3. Otherwise the new solution is stored as a novel experience.

4.3.4 Implementation of Similarity and other Functions

Functions discussed in chapter 3 are implemented using Java. Math.abs (to calculate absolute value) was very useful in Java for coding most of these functions. Most variables are defined as Floating-point as most of these functions use and return values, which range on interval (0.0 ... 1.0).

In summary, the framework discussed in chapter 3 can be implemented on a large scale, provided a flexible built-in environment is available for agents to operate. Implementing Case-based reasoning techniques are straightforward, owing to the clear-cut conceptual model described in chapter 3. Implementation of Similarity functions and other functions is also trivial as most of them are straightforward mathematical equations.

Chapter 5

Implementation Details

In previous chapter, we discussed the design and important implementation issues in building the proposed framework. This chapter discusses the actual implementation details and challenges encountered in implementing this framework.

This framework is implemented on a single sequential machine with approximately 3200 lines of code (refer to Appendix E for source code with documentation). The language of implementation is Java. We implemented this framework using RMI (Remote Method Invocation), with a plan to extend this work to distributed environments in future.

5.1 Implementation of Environment

The functionality of environment referred in section 4.1 of chapter 4 is implemented in class ApplicationMediator. Class ApplicationMediator implements interface Imediator. ApplicationMediator has methods *setBuyer* and *setSeller* for registering in the environment, *removeBuyer* and *removeSeller* methods for disposing the agents from the environment. Method *getCount* in ApplicationMediator returns number of buyers and sellers, which is used to calculate SupplyDemandRatio. Method *isSellerfound* is invoked whenever a buyer or seller registers for the same product.

5.2 Implementation of Ontology

The functionality of the ontology discussed in section 4.3.2 of chapter 4 is implemented in class Ontology. The database used to store the domain specific knowledge and conceptual similarity is modeled in Oracle Database. Public price ranges from the ontology are retrieved using method *getProdDet*. Connection to Oracle database is established using JDBC Bridge.

Method *productSimilarity* returns the similarity between two products. Method *RetSimProducts* uses *productSimilarity* and returns a vector of similar products. Class Ontology uses objects Concepsim, Product to store details about conceptual similarity and product.

5.3 Implementing Retrieval of relevant experience

Casebase in this implementation is modeled in Oracle Database as discussed in section 4.3.3.1 of chapter 4. We created two separate Casebases, one for storing buyers experience and other to store seller's experience. They are indexed on product name.

Class Case returns best relevant case from the Casebase for a given situation. Case uses class Ontology to get list of similar products, stored in Vector Simproducts. Method *getBestCase* in Case returns the best relevant case as a string. Method *getBestCase* makes connection to the Casebase and retrieves a list of cases for each product and matches the similarity between the cases. Method *attitudeSimilarity* and *rangeSimilarity* returns the similarity between two attitudes and price ranges respectively, as discussed in the chapter 3.

Method *updateCasebase* compares the similarity between the current case and the cases in Casebase and based on the match threshold it either updates the existing case or stores the case as a new case.

5.4 Implementing the functionality of Buyer and Seller Agents

Functionality of Buyer and Seller agents is implemented by class Buyer and Seller respectively.

5.4.1 Functionality of Buyer

Class Buyer implements interface IBuyer. Method *setSeller*, in Buyer is invoked by the environment to inform the buyer about the entry of a seller. Then relevant experience is retrieved and price ranges are calculated using method *calculateProdInfo* in class productInfo. Before the price range is calculated, attitude is adjusted using method *adjustAttitude* in class productInfo. Method *acceptProduct* is invoked by the clone to inform the buyer about the deal done successfully. Method *rejectProduct* is invoked by the clone to inform the buyer about the deal done successfully. These two methods maintain the communication between the clones and the parent. All the above methods are synchronized. Method *UpDateDataBase* is invoked to update the Casebase. Method *UpDateDataBase* has functionality of Merge discussed in section 3.6.3.2 of chapter 3.

5.4.2 Functionality of Seller

Class Seller implements interface ISeller. Method *setBuyerBroker* in Seller is invoked by the buyer clone to request negotiation thread. Rest of the functionality is similar to the methods discussed in the functionality of Buyer above.

5.5 Implementing Functionality of Clones

Functionality of Buyer clones and Seller clones is implemented by class BuyerBroker and SellerBroker respectively.

5.5.1 Functionality of Buyer Clone

Class BuyerBroker implements interface IBuyerBroker. Method *setSeller* is invoked by seller clone accepting to negotiate. Method *setSellerPrice* checks the price range of buyer and calculated next offer. Method *calculateAlphaprime* calculates bid increments. Method *doNegotiation* implements the offer exchanges between the clones. Method *accept* is invoked by seller clone to inform that the price is accepted. Method *reject* informs the failure of a negotiation.

5.5.1 Functionality of Seller Clone

Class SellerBroker implements interface IsellerBroker. Method *setBuyerPrice* checks the price range of buyer and calculated next offer. Rest of the functionality of Seller clone is similar to the functionality of the buyer clone above.

5.6 Challenges in Implementation

One of the tedious tasks in this implementation is book keeping, when attempted to simulate the cloning of the agents. There is lot of communication between the agents and their clones during the negotiation process. It added considerable complexity, and clone synchronization was another issue that posed complications. It is necessary that both buying and selling agents keep track of the information of all the clones involved in negotiation. We encountered situations that led to some conceptual questions like "which clone experience the buyer should store, if all the clones returned a failed

negotiation in a given situation?". In this case we stored the failure experience that took less number of exchanges.

Another issue of concern is supplying the domain knowledge for the ontology. As this implementation used a very limited ontology for testing purposes, the implemented database was sufficient to model it. But in case of large-scale implementations involving many products, supplying domain knowledge could be a tedious task, as domain knowledge is dynamic and it needs to be updated from time to time.

This framework is efficient, provided there are sufficient number cases in the Casebase to address various situations. Populating Casebase with Cases to address various situations is another challenging task in modeling the suggested framework. If there are not enough cases in the Casebase to meet all the possible situations, these agents have to rely on predefined strategies.

Another issue of concern in both conceptual and implementation point of view is that assigning weights to various factors in matching cases. Currently this framework assumes equal weight on all attributes in matching cases. Though it seems to be a reasonable measure, more need to be done in finding out the accurate weights one need to assign to the factors in the model, so that this weighting scheme would represent human thinking in buying and make these artificial agents successful in their negotiations. In adjusting α, we assumed various weights on the factors. This weighting scheme needs more investigation, as it is domain dependent.

Other software engineering challenges in this implementation include, concurrency issues. When we simulated multiple interactions between buyers and sellers, we

encountered concurrent access problem in the database, which is specifically called "Function sequence Error". This is a very rare occurrence in JDBC connectivity; the reason we figured out was concurrent accesses to the database using RMI, as the agents are trying to access the database concurrently.

Finally, we need to mention about the challenge in testing this model. We had to create different sets of buying and selling scenarios and ask people how they would react in those situations and then we used that data to compare with what the agents are doing in this framework.

Chapter 6

Evaluation of the Proposed Framework

This chapter discusses the evaluation phase of the proposed framework. The proposed research methodology in this work defines and formalizes few interesting aspects like usage of experience and attitude in human buying behavior.

We empirically evaluate the parameters and similarity functions considered in this framework with the final aim of testing the performance of this framework to incorporate various parameters and negotiation strategies in future work. First, we investigate and analyze the behavior of the agents in our framework. This analysis will lay foundation for subsequent experimental analysis of results.

The experiments involve selecting different buying and selling scenarios and observing the behavior of agents in these situations. These scenarios are classified in to two categories. Category one includes scenarios that represent variety of experiences of these agents and category two includes scenarios that represent degree of interactions. In both scenarios the attitude of agents is crucial, as it turns out to be the issue that determines success or failure of the agents in negotiations.

Section 6.1 analyzes the behavior of the agents in the proposed framework, section 6.2 describes the scenarios considered for experiments, and section 6.3 describes the results obtained from the experiments and computational issues involved.

6.1 Behavior of agents in the Framework

The two major issues in this research work are representing the *attitude* and the *experience* of agents in negotiation. The behavior of these agents is modeled based on these two factors exclusively. One might question the rationality behind modeling their behavior based on these two issues. We believe that human buying behavior heavily depends on attitude and experience. However, many other factors may affect the human buying behavior.

As discussed in section 3.4.1 of chapter3, the attitude of the agents is modeled representing the attributes like time, price and commitment for the purchase. We believe that these three attributes are very crucial in human buying behavior and they affect the purchasing decisions of humans in real life.

Another interesting aspect in this framework is the use of ontology. Before making a decision on purchasing a product, humans tend to do some background work to obtain information about the product. This background work may include knowing details of product in general by window-shopping, through ads, or from friends. Since the agents in this model do not have these facilities, they rely on the ontology to get the public price range, so that they make decisions bounded by that price range. Thus ontology supplies the required world knowledge for the agents in this model. Ideally, these agents should consult a Recommender agent, so that they get all necessary information from Recommenders [36].

The next important aspect of this framework is the use of previous experiences by these agents. Humans make purchasing decisions based on their previous experience.

Humans tend to learn from both successful and unsuccessful experiences. In case of using a successful experience they tend to follow the same strategy, they used before. Whereas in using an unsuccessful experience, they refine the strategy to improve it so that the result is a success. Humans also refine their memory of experiences by merging two similar experiences. Similarly, agents in this framework use their experience to adjust their attitude. The "learning from experience" mentioned in section 3.6.2.1 of chapter 3 is still in preliminary stage in this framework. One needs to consider more cases and different learning strategies to incorporate in this framework.

The agents in this framework use both successful and unsuccessful experiences as discussed in section 3.6.2 of chapter3. They update their experiences using the *Merge* function, suggested in section 3.6.3.2 of chapter3. The validity of the suggested merge function needs to be thoroughly investigated and plausibility of alternative functions or approaches to refine experiences need to be explored.

Finally, we need to mention the conceptual gain achieved by the creation of clones. Creation of clones helps the agents to negotiate simultaneously with many opponents. When time is important, if any of the clones finish negotiation with a success the parent agent would accept that deal, and disposes the other clones, which are still negotiating. Otherwise, the agent would wait till all the clones return the result and chooses the best deal. Similarly, when humans are involved in many negotiations if time is important for them, they tend to accept the deal that comes first. Otherwise, they tend to wait for the best possible deal.

The analysis of this framework discussed so far in this section suggests that the proposed framework handles some of the crucial issues involved in human buying

behavior in terms of representing attitudes, using world knowledge, using previous experience, and many-many negotiations. In the following section we describe different scenarios considered to simulate the affect of the above-discussed issues in experiments.

6.2 Scenarios considered for experiments

Evaluation of this framework is complex as there are many parameters to be considered. To our knowledge, there are no published results available similar to this framework. The experiments chosen to test this framework should reflect the simulation of agent's behavior, so that one could evaluate the performance of this framework in comparison with some real world buying and selling situations. It is difficult to judge "when the agents are learning" in this framework, as learning means getting better (in this case, making more successful deals) in AI. Illustrating a criterion for a successful deal is very hard as it depends on many parameters and also on the individual rationality of the evaluator of the deal. There is a need for some utility functions, which could evaluate the goodness of the deal.

The focus of this framework is, 'how the agents behave in familiar situations (similar situations encountered earlier) and in multiple interactions' using their experience and attitude. Testing this issue in experiments needs careful selection of buying and selling scenarios and cases, so that they satisfy constraints in the proposed model and also reflect real world situations. We consider two categories of scenarios in our experiments towards the aim of testing the behavior of these agents in making deals.

6.2.1 Scenarios with various experiences

Experience of these agents is an important factor in making deals as discussed in section 6.1. In testing the use of experience by these agents, one should consider different experiences of these agents. The agents in this framework tend to use both successful and unsuccessful experiences (though the use experience is limited currently in this framework). Moreover, in purchasing a product, these agents use the experience of purchasing another similar product. The scenarios considered in experiments should address all these situations to evaluate the effectiveness of this framework. Following are the scenarios considered in experiments to evaluate this model:

i) Using a successful experience of buying or selling the same product

ii) Using an unsuccessful experience of buying or selling the same product

iii) Using a successful experience of buying or selling a similar product, not the same product in question

iv) Using an unsuccessful experience of buying or selling a similar product, not the same product in question

We believe that above scenarios address most of the possible combinations experienced by humans in making decisions. Experiments involving these combinations are good measure of evaluating the behavior of agents in this framework.

6.2.2 Scenarios with various degrees of interaction (number of parties involved)

The decisions of the parties in negotiation are affected by number of parties involved. Degree of interaction is an important factor that affects agent behavior with respect to their attitude. When time is important for an agent, involving in negotiation with multiple parties would result in achieving its goal faster, as the agent has more choices

to make in the given problem space. The experiments conducted consider degree of interactions, to test the behavior of agents in various degrees of interaction. The degree of interactions considered in this framework for experiments are as follows:

i) One buyer and one seller

ii) One buyer and many sellers

iii) Many buyers and one seller

iv) Many buyers and many sellers

Capturing the behavior of these agents in experiments with the above interactions is a good measure of testing, as these interactions are most plausible situations in human buying behavior.

6.3 Experiments and Results

The experiments considered here relate to two important issues on which the human buying behavior is dependent in general: i) scenarios with various experiences and ii) scenarios with various degrees of interaction as discussed in previous section.

6.3.1 Experiments testing scenarios with various experiences

Our assumption about the affect of various experiences in negotiation could be stated as follows:

> *"Agents with relevant experience in buying or selling should be able to use that experience in buying and selling similar products"*

To evaluate this assumption we need to provide the agents with different types of experiences for various ranges of products. These agents use their relevant previous experience in buying and selling similar products. The results of experiments as shown in Appendix D, suggests that they pick up the best possible experience in solving a

similar problem. We observe that match threshold over 0.75 seems to be reasonable measure in matching cases. The weighting scheme used in *Match* function explains the reasons behind this observation. Note that Match function assigns equal weight on each of the attributes, Product similarity (PS), Range Similarity (RS), and Attitude Similarity (AS). Attitude and Price are the most important attributes in purchasing decisions. Both PS and RS very much dependent on the price of the product and AS completely depends on Attitudes. If any of the attributes in Match function differ drastically, then the total weight returned by match function is less than 0.75. Therefore a match threshold over 0.75 seems to be a reasonable measure in choosing right experience to use in making deals. If the match threshold is less than 0.75 the agents should fall back on using some predefined strategies.

The assumption that *"Agents with relevant experience in buying or selling should be able to use that experience in buying and selling similar products"* is valid as the above discussion supports it. The results from the experiments also support that these agents use both successful and unsuccessful experiences in making deals. The results also suggest that these agents update (refine) their experiences. However, we admit that the merge function in this framework is a simple idea to update the experiences. There is a need for sophisticated learning approaches and functions to update the Casebase.

6.3.2 Experiments with various degrees of interaction

Our assumption about the affect of Degree of Interaction on negotiation could be stated as follows:

> *" Many to Many interactions may help the agents to finish negotiations faster".*

To evaluate the above assumption we need to simulate situations discussed in section 6.2.2 of this chapter. The experiments conducted represent all the situations discussed towards the aim of determining agents behavior in this framework.

Considering the above assumption we observe that the agents in this framework tend to make deals at a faster rate when they are involved in multiple negotiations. When there are one-one negotiations these agents don't have many choices, so that they continued negotiation until an agreement is reached or they abandon the negotiation if their interests are not met. Whereas in case of many to many interactions agents have more choices and they tend to be more selective in making deals, for their benefit.

Another interesting observation we made in the experiments that those agents who were successful in a one-one negotiation with a given attitude and experience, failed to make deals in many-many interactions with same attitude and experience, owing to the behavior of other agents in the market. The agents with high importance of time and less importance of price in attitude tend to pocket the advantage of many-many interactions.

The observations made out of these experiments infer that the agents in this framework use similar experiences in finding solution for a new problem, tend to refine their experiences and many to many interactions may help them to finish the negotiations at faster rate.

This framework handles some of the important issues discussed in section 1.3.1 of chapter 1. Agent interactions in this framework address the issue of cardinality of negotiation, as this framework handles all possible interactions (one to one, one to many and many to many). Agent characteristics like commitments, goals are taken in

to consideration in this framework. This framework also addresses public and private value of goods, as it represents public price range and agent price ranges. Event parameters like importance of Time, timeouts are also considered in this framework. This framework handles information parameters with the use ontology and Casebase to provide the agents with domain specific knowledge and experience respectively. Thus this framework addresses some of the issues related to individual rationality in negotiations. The computational issues involved in building this framework are discussed in the following section.

6.4 Computational Issues

The general properties desirable for a negotiation mechanism are computational efficiency, communication efficiency and distribution of computation as mentioned in section 1.3.1 of chapter 1.

6.4.1 Computational Efficiency

A negotiation mechanism has to be computationally efficient. The computational complexity in executing strategies needs to be carried out at run-time. Therefore it should be manageable. We analyze the complexity of this framework in two phases. In phase1 we consider the complexity in retrieval of information from the ontology and retrieval of experience from Casebase. Phase 2 considers the complexity in executing the strategy in various interactions of these agents.

Retrieval of information from ontology is straightforward in this implementation as the ontology is represented in a database and it takes cost of an SQL Query. However, in case of a large ontology using formal approaches, the complexity in retrieving

information is proportional to the depth of the hierarchy and also the number links to be accessed.

In this implementation, the complexity involved in retrieving cases is linearly proportional to number of relevant cases in the Casebase, as the number of cases considered is very limited. However, in case of large Casebase with thousands of cases, efficiency of the system depends on proper indexing of cases, which is one of the most important issues in Case-based reasoning.

In phase 2, we analyze the complexity involved in different interactions of the agents in this framework. Our implementation and experimentation is done on a sequential machine. The complexity of the whole process of negotiation is '$m \times n$', where 'm' is number of unique buyers and 'n' is number of unique sellers for the same product in the marketplace. Each buyer creates as many number of clones as the number of unique sellers in the market. For example, if there are 2 buyers and 3 sellers in the marketplace, 3 clones are created by each buyer. The exchange of offers and counter offers is done in constant time, as there is no significant computation involved. However, this could get really complex on incorporation of different strategies in the framework.

In case of a distributed environment the complexity is linear with respect to the number of relevant sellers. It takes linear time to create the clones by the buyer. Then these clones go on different machines and negotiate with the sellers on those machines. Exchange of offers and counter offers is constant. Therefore the whole process is linear when operated in parallel.

6.4.2 Communication Efficiency

The communication between agents in this framework is facilitated by RMI (Remote Method Invocation). It is secure to use RMI as clients can invoke only the provided methods in the interface. Another advantage of using RMI is that users interacting with this application need only those methods defined in the interface.

However, in a distributed scenario there is a need for more sophisticated protocols to facilitate communication between the agents and clones. Agent communication languages like KQML and ACL may come handy in implementing distributed heterogeneous systems.

6.4.3 Distribution of computation

Currently, this framework does not handle distribution of computation. This framework may be extended to operate in distributed environments on incorporation of more sophisticated protocols. The major concern in extending this framework for a distributed environment is "mobility of code". This issue could be addressed by reframing this framework on an environment like IBM Aglets, which supports issues like code mobility.

Another important issue is control of clones in a distributed scenario. One suggestion to handle this issue is to have a centralized control over clones. Parents have a handle on each clone and the clones in turn return the result of the negotiation to parents. Whenever a clone does not return a result in the specified time, the clone will be timed out of the negotiation after the specified time interval. This centralized control over clones avoid inconsistencies arise out of concurrency and also avoid stalemates in negotiations where price is important for agents.

6.4 Critical Analysis of the Framework

Evaluation of this framework suggests that it is a reasonable effort in automating agent negotiations. In this section we address the issues overlooked in modeling this framework. This will lay the foundation for scope of future work in this framework.

6.4.1 Limitations

This framework is very simple at present, not representing many aspects of human buying behavior. At this stage the framework supports only single environment in terms of agent interactions. But this could be extended to a distributed environment, where agents from different markets could negotiate in parallel. This adds considerable complexity to the framework and raises other questions like security, compatibility, and consistency.

Another limitation is that we are using only one best experience in adjusting attitude, calculating agent price range and bid increments. But using multiple experiences to solve current situation would open more avenues for the agents to act up on. Using a set of positive and negative experiences to act up on a situation generates more choices for agents in learning and adjusting their attitude and bid increments. This issue needs serious consideration in future work.

This framework does not represent all issues involved in human buying behavior, though it represents human buying behavior to certain extent. For example if a buyer is involved in the following situation:

'It is very likely that the price of PCs will keep going down for a while'
Buyer can wait for another few months to buy a PC.

Then,

The buyer should probably wait a few months to buy a PC, unless he stumbles on a very good deal.

The proposed framework is too far away from formalizing similar type of reasoning. Some commonsense reasoning strategies need to be incorporated in the framework to address temporal and modal aspects.

6.4.2 Compromises

In modeling and implementation we took certain shortcuts and made some compromises to reduce complexity of the framework. One of the shortcuts includes modeling of ontology. In the implementation, we assumed very limited domain specific knowledge with ontology being just one level. The ontology should be considerably extended to represent more domains to test the behavior of these agents in different domains.

Process of updating the case base is another issue of concern, leaving us with some conceptual and software engineering questions. For example, there is one buyer and many sellers in the market. Imagine a situation, where the buyer created as many clones with respect to number of sellers and all the clones returned a failed negotiation to the buyer. The question here is, which experience the buyer should store? For now, we are storing the negotiation that took less number of exchanges as buyer's new experience. Validity of this reasoning needs to be thoroughly tested using different strategies to store experience.

In modeling human buying behavior supply demand ratio is not the only external factor of importance. Economic factors like inflation, recession and other issues are not

considered in this framework. Future work could be extended in investigating plausibility of incorporating these issues in the framework.

Other issues that we overlooked in this work and could be extended in future work include, representing the experience personal to a specific agent exclusively, instead of current practice in this framework, representing it as public experience. This leads to very interesting situations and the behavior of agents would become very specific reflecting their own personal experience. Another issue that could be considered is storing the opponent agent's profile when storing experience. This could help agents to choose opponents in negotiation based on the number of success with them in their previous negotiations.

6.4.3 Scalability issues

The examples tested in this framework are limited as shown in appendix D. To make this framework a reality, Casebase should have thousands of cases. Then it would be possible to test more interesting scenarios for different domains. In this framework, we assumed very limited ontology and limited amount of cases for testing purposes. When this framework is used for real applications huge amount of data is needed in Casebase for these agents to reason and learn efficiently. Increasing the size (levels) of ontology assumed is another issue needs to be addressed in future work.

As it is discussed in this subsection, there is lot of scope for improving this framework in future work. In summary, the important issues one needs to consider for future work in this framework include the following:

- Trying out different weighting schemes on the attributes in matching cases
- Increasing the size of domain specific knowledge (ontology)

- Using more than one case in learning the optimal attitude and calculating new bids
- Incorporating more economic factors and parameters in the framework

Chapter 7

Conclusion and Future Work

7.1 Framework to Automate Negotiations

In this thesis, we proposed a framework to automate negotiation among software agents in virtual marketplaces. The purpose of this work is to investigate the thesis, possibility of building a framework which models limited aspects of negotiation among software agents.

These agents in this framework have attitude that represents few aspects of mental state of its client when the agent is created. Agents in this framework learn from experience (with respect to Case-based Reasoning) and do simple reasoning to make smart choices by adjusting their attitude and their bid increments. The interaction among these competing agents is many to many. They also tend to refine their previous experiences in connection with new situations they encounter from time to time.

The proposed framework is implemented in a non-distributive environment to test the behaviour of the agents. Java is used as the language of implementation. The following is a summary of the work that has been done:

- Designed a prototype for implementation of virtual marketplace that contains a Casebase and Ontology for domain specific knowledge.
- Represented the attitude of these agents using three attributes with fuzzy values, which is a novel approach in this work.
- Limited Learning (in the sense of Case-based Reasoning) in these agents is modeled by using Case Based Reasoning techniques.

- This framework biases its agents towards making a successful negotiation, by adjusting the attitude, which aims at a successful negotiation.
- Many to Many interactions are simulated among these agents in the experiments.

7.2 Analysis of Experiments

In this work we built a framework to automate negotiation among software agents in virtual marketplace, to investigate the following thesis:

"It is possible to build autonomous agent systems which model limited (important) aspects of negotiation using current software technologies"

As we have discussed in chapter 6, the agents in this framework represent some important aspects of mental state of their client and learn limitedly from experience (in the sense of Case-based Reasoning). The experiments further demonstrated that many to many interactions might help the agents to finish negotiations at faster rate as they have more choices.

The conclusions drawn from the experiments can be summarized as follows:

- It is highly possible to build a framework where limited aspects of negotiations among software agents are modeled.
- It is possible represent some important aspects of mental state of the clients in automated negotiations involving software agents.

- The proposed framework simulates interaction of mental state and experience
- The experiments also suggest that these agents are learning limitedly from both successful and unsuccessful experiences (increased cases in the Casebase).
- The modifications to cases in the Casebase done by these agents during experiments suggest that they are refining their experiences.
- Final point has to be made about Many-Many interactions as they might help agents finish negotiations at a faster rate.

7.3 Future work

The critical analysis of the framework in section 6.4 of chapter 6 suggests that there is lot of scope for future work in the proposed work.

Currently the framework is too simplistic, further extensions are needed to upgrade this framework considering more parameters like beliefs, predictions and more economic factors. Some form of uncertainty reasoning must be incorporated in to the framework since market conditions and agent's beliefs are dynamic and uncertain, and are rarely crisply defined.

The weighting scheme discussed in chapter 3 for various similarity functions needs thorough investigation to find out the optimal weighting scheme, perhaps conducting a machine learning experiment in future extensions of this work.

Another issue, which is left for future is that considering more than one case in adjusting attitude of the agents. Considering a set of successful and unsuccessful cases in adjusting attitude may be ideal, as the agents will have more examples to learn from. One need to give serious thought about this observation, as this kind of extension could really simulate the way humans think in purchasing decisions.

We need to incorporate different strategies in this framework to model more aspects of human buying behaviour. Different strategies from Game Theory and Economics need to be considered in future work, to incorporate them in this framework.

Making the experience exclusively personal for each agent and maintaining a list of reputable[12] counterparts for negotiation are other issues to be investigated in future work, in modeling these agents to behave like humans in virtual marketplaces.

---

[12] Reputable opponents are the agents with whom this agent had successful negotiations in the past. Maintaining a list of reputable opponents helps the agent to give preference to the agents they already know.

Bibliography

1. Abdel-Illah Mouaddib. 1997. **Progressive Negotiation For Time-Constrained autonomous Agents**. *Agents'97 Conference Proceedings, ACM.*

2. Alexandros Moukas, Robert Guttman , Pattie Maes. 1998. **Agent-mediated Electronic Commerce: An MIT Media Laboratory Perspective**. *Proceedings of the International Conference on Electronic Commerce.*

3. A.Chavez, D. Dreilinger, R. Guttman, and P. Maes. 1997. **A Real-Life Experiment in Creating an Agent Marketplace**. *Proceedings of the Second International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology (London UK).*

4. A. Chavez and P. Maes. 1996. **Kasbah: An** Agent **Marketplace for Buying and Selling Goods**. *Proceedings of the First International Conference on the Practical Application of* Intelligent *Agents and Multi-Agent Technology (London, UK).*

5. C. B. Excelente-Toledo, R. A. Bourne and N. R. Jennings 2001. **Reasoning about commitments and penalties for coordination between autonomous agents**. *Proc. 5th Int Conf on Autonomous Agents (Agents-2001), Montreal, Canada.*

6. Mudgal, J. Vassileva 2000. **Multi-agentnegotiation to support an economy for online help and tutoring**. *Proceedings of ITS'2000, Springer LNCS 1839, 83-92.*

7. Daniel M Reeves, Benjamin N Grosof, Michael P Wellman, Hoi Y Chan. 2000. **Automated Negotiations from Formal Contract Descriptions**. *IBM/IAC Workshop on Internet-Based Negotiation Technologies.* AAAI.

8. E. Oliveira, J. M. Fonseca, N. R. Jennings. 1999. **Learning to be Competitive in the Market**. *AAAI Workshop on Negotiation: Settling Conflicts and Identifying Opportunities(Orlando, FL, 30-37).*

9. F. Nolan, J. Wilkiewicz, D. Dasgupta and S. Franklin. 1999. **Evolutionary Economic Agents**. *Conference on American Association for Artificial Intelligence (AAAI).*

10. Hayes-Roth, B. 1995. **An Architecture for Adaptive Intelligent Systems**. *Artificial Intelligence: Special Issue on Agents and Interactivity, 72, 329-365.*

11. Hyacinth S. Nwana. 1996. **Software Agents: An Overview.** *Knowledg Engineering Review, Vol. 11, No 3, pp. 1-40, Sept 1996.*

12. Hyacinth S. Nwana, Divine T. Ndumu. 1999. **A Perspective on Software Agents Research.** *The Knowledge Engineering Review, Vol 14, No 2, pp. 1-18.*

13. Jae-Yeon Kang, Eun-Seok Lee.1998. **A Negotiation Model in Electronic Commerce to Reflect Multiple Transaction Factors and Learning.** *Proceedings of the 13th International Conference on Information Networking (ICOIN '98).*

14. Janet Kolodner. 1994. **Case-Based Reasoning**. *Morgan Kaufmann Publishers. ISBN 1-55860-237-2, Artificial Intelligence.*

15. Jeffery O. Kephart , Amy R. Greenwald 1999. **Shopbot Economics**. *Lecture Notes in AI vol.1638.*

16. L. Esmahi and P. Dini, J.C. Bernard. **Toward an Open Virtual Market Place for Mobile Agents**. *Proceedings of the IEEE 8th International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises.*

17. M.T. Tu, C. Seebode and W. Lamersdorf. 1998. **A Dynamic Negotiation Framework for Mobile Agents.** *Proceedings of the First International Symposium on Agent Systems and Applications Third International Symposium on Mobile Agents.*

18. Manoj Kumar, Stuart I. Feldman. 1998. **Business Negotiations on the Internet.** *Technical Papers, IBM Institute of Advanced Commerce. http://www.ibm.com/iac/reports-technical/reports-bus-neg-internet.html*

19. Mario Lenz, Brigitte Bartsch-Sporl, Hans-Dieter Burkhard, Stefan Wess. **Case-Based Reasoning Technology From Foundations to Applications**. *Lecture Notes in Artifcial Intelligence 1400. ISBN 3-540-64572-1.*

20. Mark E. Nissen and Anshu Mehra. 1999. **Some intelligent software supply chain agents.** *Proceedings of the third annual conference on Autonomous Agents, 1999, Pages 374 –375. ACM.*

21. Micheal R. Genesereth. 1994. **Software Agents**. *Communications of the ACM, july 1994/vol. 37, No.7.*

22. M. Wooldridge and N.R. Jennings. 1995. **Intelligent Agents: Theory and Practice.** *The Knowledge Engineering Review, 10 (2), pp. 115-152, 1995.*

23. N. R. Jennings. 1999. **Agent-Oriented Software Engineering.** *Proceedings of the 12th Int Conference on Industrial and Engineering Applications of AI, Cairo, Egypt, 4-10.*

24. Nicholas R. Jennings · Michael J. Wooldridge. 1998. **Agent Technology: Foundations, Applications, and Markets.** *Publication Date: February 25th, 1998. Publishers. Springer and Unicom. ISBN: 3540635912.*

25. N. R. Jennings, S. Parsons, C. Sierra and P. Faratin. 2000. **Automated Negotiation.** *Proceedings of the 5th International Conference on the Practical Application of Intelligent Agents and Multi- Agent Systems (PAAM-2000), Manchester, UK, 23-30.*

26. N. R. Jennings, P. Faratin, T. J. Norman, P. O'Brien and B. Odgers. 2000. **Autonomous Agents for Business Process Management.** *International Journal of Applied Artificial Intelligence 14 (2) 145-189.*

27. N. R. Jennings, P. Faratin, A. R. Lomuscio, S. Parsons, C. Sierra and M. Wooldridge 2001. **Automated negotiation: prospects, methods and challenges.** *Int. J. of Group Decision and Negotiation 10 (2) 199-215.*

28. P. Faratin, C. Sierra, N. R. Jennings and P. Buckle. 1999. **Designing Responsive and Deliberative Automated Negotiators.** *Proceedings of the AAAI Workshop on Negotiation: Settling Conflicts and Identifying Opportunities, Orlando, FL, 12-18.*

29. P. Faratin, C. Sierra and N. R. Jennings. 2000. **Using similarity criteria to make negotiation trade-offs.** *Proceedings of the 4th International Conference on Multi-Agent Systems (ICMAS-2000), Boston, USA, 119-126.*

30. Pattie Maes. 1994. **Agents that reduce work and information overload.** *Communications of the ACM. Volume 37, No. 7 (Jul. 1994).*

31. Pattie Maes, Robert H. Guttman and Alexandros G. Moukas. **Agents that buy and sell.** *Communications of the ACM, Volume 42, No. 3 (Mar. 1999).*

32. R. Guttman and P. Maes. 1998. **Cooperative vs. Competitive Multi-Agent Negotiations in Retail Electronic Commerce.** *Proceedings of the Second International Workshop on Cooperative Information Agents (CIA'98), Paris, France, July 3-8, 1998.*

33. R. Guttman, P. Maes, A. Chavez, and D. Dreilinger. 1997. **Results from a Multi-Agent Electronic Marketplace Experiment.** *Proceedings of Modeling*

Autonomous Agents in a Multi-Agent World (MAAMAW'97), Ronneby, Sweden, May 1997.

34. Sandip Sen, Partha Sarathi Dutta, Rajatish Mukherjee. 2000. **Agents that represent buyer's interests in E-commerce.** *AAAI 2000 KBEM Workshop - July 30-31, Austin.*

35. Stan Franklin, Art Graesser. 1996. **Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents.** *Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages, Springer-Verlag, 1996.*
36. Thomas Tran, Robin Cohen. 1999. **Hybrid Recommender Systems for Electronic Commerce.** *Knowledge-based Electronic Markets a AAAI'00 Workshop (KBEM'00) Monday, July 31, Austin TX, USA.*

37. Sandholm, T. and Vulkan, N. 1999. **Bargaining with Deadlines**. *National Conference on Artificial Intelligence (AAAI), pp. 44-51, Orlando, FL.*

38. Vulkan N., Binmore K. *1997.* **Applying game theory to automated negotiation**. *DIMACS Workshop on Economics, Game Theroy and the Internet, 1997.*

39. W.Y.Wong, D.M.Zhang, and M. Kara-Ali. 2000. **Towards an experience-based negotiation agent**. *Proceedings of the 4th International Workshop on Cooperative Information Agents, CIA-2000, Boston.*

40. F. Zambonelli, N. R. Jennings, A. Omicini and M. Wooldridge 2001. ***Agent-Oriented Software Engineering for Internet Applications.*** Coordination of Internet Agents (eds. A. Omicini, F. Zambonelli, M. Klusch and R. Tolksdorf) Springer Verlag, 326-346.

Appendix A Software Agent Technology

A.1 Software Agent technology

Software is very diverse. Even though programs provide users with significant value, *interoperability* is a major problem and *Heterogeneity* is the cause of problems in *interoperation* [21].

Agent based computing is a very recent approach to problem solving in complex heterogeneous systems. These systems are called "Intelligent Software Agents" by the AI community, and in particular, the Agent Community. Agent-based software engineering came about to facilitate the creation of interoperable software in such settings where there is heterogeneity. In case of interdependent problems between the modular components, agents cooperate with each other to ensure that the interdependencies are properly managed. Agent technology provides us with inherently distributed data, control and resources. These agents can perform different tasks like managing information by gathering the information and filtering the information as per user needs [30], playing an important role in electronic commerce by buying and selling goods [31], managing business process management by handling things like supply chain management [20], and being helpful in health care by patient condition monitoring. Therefore Agent-based computing has the potential to significantly improve the theory and practice of modeling, designing, and implementing complex systems.

The research on intelligent software agents has been going for about 15 years. The word 'Agents' really became popular in computer magazines and journals around 1995.

The concept of a software agent can be tracked back to the early days of the research into Distributed Artificial Intelligence (DAI) in the 1970's. The Actor model proposed by Hewitt in 1977 was a concept of a self-contained, interactive and concurrently executing object, which he termed eactori. This object had some encapsulated internal state and could respond to messages from other similar objects. : An actor. The evolution of agent technology is illustrated in figure A.1.

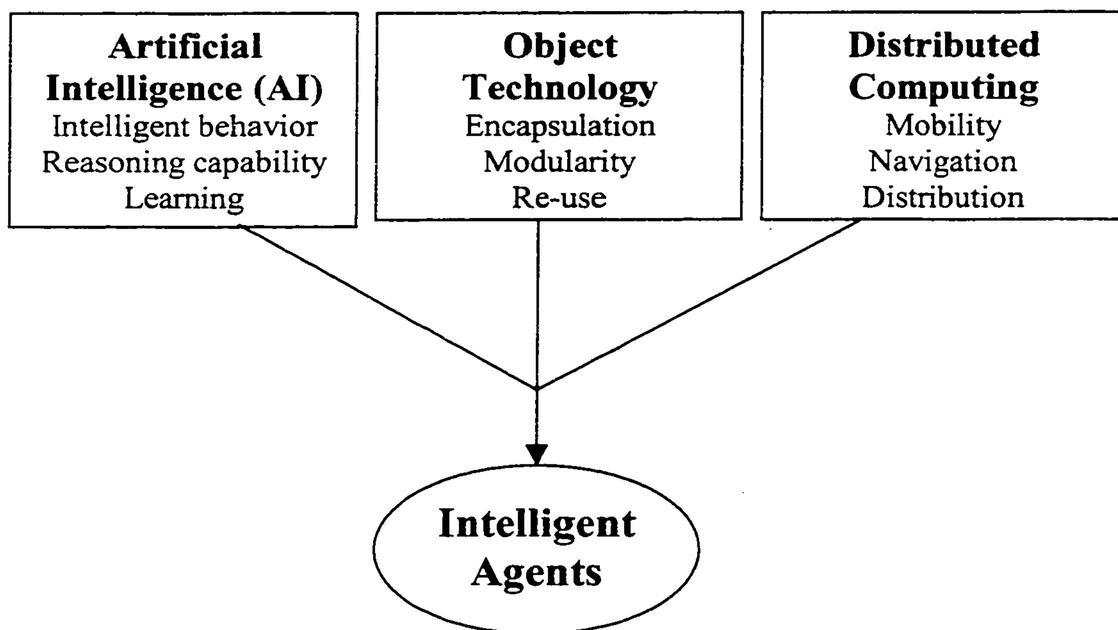


Figure A.1: Evolution of Agent technology

A.2 What is an agent?

Though there is an understanding about the evolution of Software agents, agent based computing lacks universally accepted definitions. It is difficult to precisely define "what an agent is", as agent is an umbrella term for a heterogeneous body of research and development in computer science. The confusion about agents led researchers to invent more synonyms including know bots (knowledge-based bots), soft bots

(software robot), task bots (task-based robots), user bots, robots, personal agents, autonomous agents (mobile agents), auction bots and personal assistants [11]. Agents have in many physical appearances: for example, those that inhabit the physical world, e.g. a factory are called *robots* that operate in dynamic and uncertain environments. The Mars polar lander is a very good example of this type. Those that inhabit vast computer networks are called soft bots. The agents that perform certain specific tasks are referred as task bots. The other kind of agents may play many roles, like search agents, report agents, presentation agents, navigation agents, role playing agents, management agents, search and retrieval agents, domain specific agents, development agents, analysis and design agents, testing agents, packaging agents and help agents [11].

The term agent is widely used by many people working in closely related fields. Therefore there is a need to define "what an agent is" to avoid misuse of this term. As per the weak notion of agency, agents are autonomous, have social ability, and they are proactive and reactive. In the stronger notion of agency, some researchers believe that agents use mentalistic notions like knowledge, belief, intention, emotion and obligation. Few researchers defined "what an agent is" consolidating the above features and properties.

According to Jennings and Woodridge [23] an agent is,

"*an encapsulated computer system that is situated in some environment, and that is capable of flexible, autonomous action in that environment in order to meet its design objectives.*"

Another definition given by Pattie Maes, MIT [35] is,

"Autonomous agents are computational systems that inhabit some complex dynamic environment, sense and act autonomously in this environment, and by doing so realize a set of goals or tasks for which they are designed."

The essence of all of the above definitions is that there is a unanimous agreement that agents are *autonomous* and are situated in an environment. For example a regular payroll program in a real world environment takes input and acts on it producing an output. This payroll program is not an agent because its output would not normally effect what it senses later. It fails the "over time" test of temporal continuity, since it runs once and goes into coma, waiting to be called again. Most of the ordinary programs are ruled out as agents, because one or both of the above conditions are not met. Therefore we can infer from the above discussion that "***All software agents are programs, but not all programs are agents*** " [35]. Therefore "a program must measure up to several marks to be an agent".

The above discussion about evolution of agent technology and notion of agency gives a brief understanding that 'what programs can be called as agents'. The next issue to address is the functionality of these agents and the environment required. This issue can be addressed by focussing on properties and classification of the agents. Agent properties are classified into two categories. *Common properties* are applicable for all agents and *classifying properties* are used to categorize different agents.

A.3 Common properties

All agents have these common properties, as these properties are essential for a program to be called an agent. As its was discussed earlier, agents are autonomous,

they are situated in an environment, and they are goal oriented. Each property is discussed in detail in the following discussion.

A.3.1 Life Cycle: An agent is created by an authority as an instance of a class through a creation operation. An agent can only be destroyed by the same authority, which created this agent, or on its behalf. An agent has a name space at any moment within which it can act. This name space consists of the names of known agents and agent systems including itself. The name space can be related to the environment the agent is currently in.

A.3.2 State, Reflexivity: An agent has to be capable of reflecting its internal process and state. This is equivalent to the necessity to represent meta-knowledge about its internal structure. The state can comprise a list of messages and statements about the state of activity of the agent.

A.3.3 Autonomy: Autonomy means an independent and decoupled execution of tasks undertaken by an agent. The interpretations of a piece of messages, orientation etc. are determined only by the agent. Its autonomy ends where its resources are depleted, and it depends on cooperation.

A.3.4 Locality: Each agent has knowledge, which it uses to fulfill its tasks. The local knowledge is determined by the agent's profile, its state, including the list of known agents. An agent initially does not know anything about the states of other agents.

A.3.5 Structural Openness: Agents can change over time and thus show a new behaviour or re-organize the structure of relations between themselves and other agents, too.

A.3.6 Authority: The authority of an agent identifies the person or the organization for which an agent is activated. The authority has to be verifiable. Agents keep their authority during the whole life cycle.

A.3.7 Security: Concerning agents authorizing, communication, coordination, mobility and consumption of resources, etc., an agent application has to meet high security demands. Agents, agent connectors and agent systems are responsible for the warranty of security properties.

A.3.8 Goal: Execution of tasks undertaken by agents are goal-oriented. To this end a schedule is drawn up. Within the scope of this schedule an agent acts actively to achieve the goals.

A.4 Classifying Properties

Apart from the above properties, software agents can have other capabilities like communication, mobility and cooperation. A combination of all these properties may not be possible, but agents can certainly have few of those abilities. These properties are used to categorize the agents. Thus these properties are called classifying properties.

A.4.1 Communication Behaviour: Each agent belonging to an instance of an agent system can communicate within its name space according to its behaviour at any moment. The behaviour of an agent determines whether it carries out tasks delegated to it in cooperation with other agents or it is capable of doing this on its own.

A.4.2 Locality Affiliation and Mobility: The locality of an agent is the location where the agent is located within the network. A mobile agent can change its place in the

system during its work. Mobile scripts can be collected in one place, moved to another place and get executed there. Mobile objects are moved from one place to another during run time and task execution. Mobility has utmost importance in these days, as distributed computing is in limelight.

A.4.3 Cooperation: Agents need to possess a social ability to interact with other agents and possibly humans via some communication language for coordination.

A.4.4 Negotiation Ability: Negotiation ability describes the properties of an agent to execute a task collaborating with other agents and to negotiate this cooperation. When the agent accepts a task, it stores it in a task list. Agents divide the work and distribute the work among themselves.

A.4.5 Delegation Ability: Agents can place and take on tasks. Delegation means that partial tasks can be passed on to agents. Agents execute these tasks according to the results they can deliver and execution control.

A.4.6 Learning Adaptability: The intelligence of an agent is the level of its evaluating and learning behaviour. An agent learns by executing tasks and uses the acquired knowledge during task execution.

A.4.7 Re-Usability: Process or subsequent instances may require to keep instances of the class 'agent' for information hand-over or to check and to analyze them according to their results. The reusability of agents can take place in various agent systems of a different authority.

A.5 Classification of agents

Agents may be classified based on the mobility factor, i.e. *static* or *mobile*. Another way of classification is either *deliberative* or *reactive*. Deliberative agents process an

internal symbolic reasoning model and they engage in planning and negotiation in order to achieve coordination with other agents. Reactive agents do not have any internal symbolic models of their environment, and they act using a stimulus or response type of behaviour by responding to the present state of environment in which they are embedded [11].

At BT labs, agents are classified based on three properties, Autonomy, Learning and Cooperation. By using these three characteristic properties they derived four types of agents viz., Collaborative agents, Collaborative-learning agents, Interface agents and truly smart agents. The following figure is taken from [11].

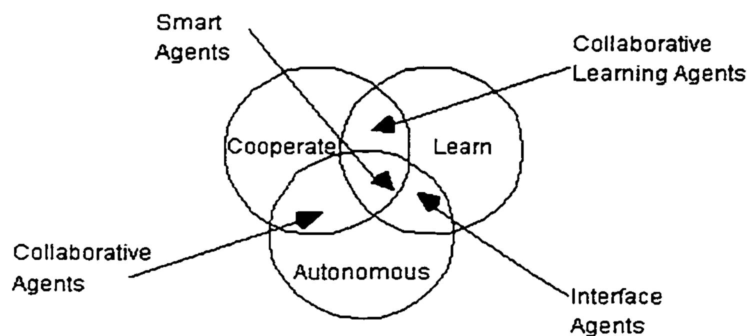


Figure A.2: Agent typology based on three properties

Agents can also be classified based on their roles. In this category, different types include information agents, Internet agents, report agents, presentation agents, analysis agents, testing agents and help agents. Another categories of agents are hybrid agents, which combine of two or more agent philosophies in a single agent. Based on the different categories of agents, discussed above, seven types agents are identified.

1. Collaborative agents
2. Interface agents
3. Mobile agents

4. Information/Internet agents
5. Reactive agents
6. Hybrid agents
7. Heterogeneous agent systems.

A.5.1 Collaborative agents:

Collaborative agents emphasize autonomy and cooperation in order to perform tasks of their owners. In order to collaborate they may have to negotiate in order to reach mutually acceptable agreements on some matters. General characteristics of these agents include autonomy, social ability, responsiveness and pro-activeness. They are able to act rationally and autonomously in open and time constrained multi agent environment. Robustness, effectiveness, scalability and maintainability of these agents can be improved engaging these agents in automated negotiations.

A.5.2 Interface agents (User agents):

These agents emphasize autonomy and learning in order to perform tasks for their users. Collaborating with user may not require an explicit agent communication language as one when collaborating with other agents. Thus the user agents act as an autonomous personal assistant, which cooperates with the user in accomplishing some task in the application.

A.5.3 Mobile agents:

Mobile agents are computational software processes, which are autonomous, cooperative and capable of roaming wide area networks (WANs) such as the WWW, interacting with foreign hosts, gathering information on behalf of its owner and coming

back to the user having performed the duties set by its user. These duties may range from flight reservation to managing a telecommunication network.

A.5.4 Information/Internet agents:

Information agents help the user manage, manipulate and collate information from many distributed sources. These agents may be static or mobile. They may be non-cooperative or social and they may learn or may not learn. Hence, there is no standard mode to their operation. The information agent may be associated with some particular indexer(s), e.g. a Spider. The user information agent, which has been requested to collate information on some subject, issues various requests to one or several URL search engines to meet the request. Some of this search may even be done locally if it has local cache. The information is collated and sent back to the user.

A.5.5 Reactive agents:

These agents do not possess internal, symbolic models of their environments, instead they respond in a stimulus-response manner to the present state of the environment in which they are embedded. A reactive agent is viewed as a collection of modules, which operate autonomously and are responsible for specific tasks. Communication between modules is minimized and is of low-level nature. The key benefits of these agents are that they would be more robust and fault tolerant than other agent based systems. A favourite application area for these agents seems to be the games or entertainment industry.

A.5.6 Hybrid agents:

Hybrid agents constitute a combination of two or more agent philosophies within a single agent. These philosophies include a mobile philosophy, an interface agent

philosophy, collaborative agent philosophy, etc. The benefits of these agents would be to set union of the benefits of the individual philosophies in the hybrid, that would make these systems robust, faster response times and adaptable.

Hayes-Roath's integrated architecture for intelligent agents [10] consists of two layers (i) The physical layer which performs perception-action coordination, i.e. it sense, interprets, filters and reacts to the dynamic environment in which the agent is embedded; and (ii) The cognitive layer receives perceptual input from the physical controller to construct an evolving model, and to perform interpretation, reasoning and planning.

A.5.7 Heterogeneous Agent Systems:

These agents are integrated set-up of at least two or more agents, which belong to different agent classes. A heterogeneous agent system may also contain one or more hybrid agents. Agent-based software engineering has been invented in order to facilitate the interoperation of miscellaneous software agents. A key requirement for interoperation amongst heterogeneous agents is having an agent communication language (ACL) via which the different software agents can communicate with each other.

Appendix B Negotiation

B.1 Negotiation and Negotiation Theory

Negotiation is a form of decision-making where two or more parties jointly search a space of possible solutions with the goal of reaching a consensus for their own benefits. Economics and Game Theory describe such an interaction in terms of protocols and strategies [32].

Another definition for automated negotiation by Jennings et al. [25] is 'the process by which a group of agents communicate with one to try and come to a mutually acceptable agreement on some matter'. Negotiation underpins attempts to cooperate and coordinate and is required both when they are self-interested and when they are cooperative. As we have discussed in the properties of agents these two properties, cooperation and coordination play major role here.

Negotiation theory covers a range of phenomena and encompasses multifarious approaches like Artificial Intelligence, Social Psychology, and Game theory. Negotiation research can be considered to deal with three broad topics [25].

1. Negotiation Protocols: These are the set of rules that govern the interaction. This includes the types of participants, the negotiation states, the events, which cause negotiation states to change, and the valid actions of the participants' in particular states.
2. Negotiation Objects: The range of issues over which agreement must be reached. At one extreme, the object may contain a single issue, while on the other hand it may cover many issues, which makes the negotiation process complex.

3. Agents' Decision Making Models: The decision making apparatus the participants employ to act in line with the negotiation protocol in order to achieve their negotiation objectives. The sophistication of the model, as well as the range of decisions which have to be made, are influenced by the protocol in place, by the nature of the negotiation object, and by the range of operations which can be performed on it.

The relative importance of these three topics may vary according to the negotiation and environmental context. Given a wide variety of possibilities, there is no universally best approach or technique for inter-agent negotiation.

The minimum capabilities required for a negotiation is: (1) to propose some part of the agreement space as being acceptable and (2) to respond to such a proposal indicating whether it is acceptable. But, if agents can only accept and reject others proposals, then negotiation can be very time consuming and inefficient. Because the proposer has no means of knowing why the proposal is unacceptable, or whether the agent is neither close to an agreement, nor in which direction of the agreement space it should move next.

B.2 Negotiation Process

To improve the efficiency of the negotiation process, the recipient needs to be able to provide more useful feedback on the proposals it receives than just whether or not it agrees to them. This feedback can take the form of a critique or a counter proposal. From such a feedback, the proposer should be able to generate a proposal, which is more likely to lead to an agreement. To do these proposals and counter proposals the agents should act intelligent with some reasoning capabilities. These agents will have

to follow different strategies and negotiation algorithms and a set of negotiation tactics. When we equip these agents with these negotiation techniques and tactics, they can negotiate without any human interaction at any places like classified negotiation, stock market negotiation and retail auction negotiation [32].

B.3 Automated Negotiation

Formalization of the above discussed negotiation process has been attracting the researchers in multi-agent systems, in the wake of exponential growth of e-commerce. In most cases negotiation strategies and tactics are completely domain dependant. There are many protocols and parameters, which are to be considered in an attempt to formalize the process of negotiation [27]. The general properties desirable for a negotiation mechanism are computational efficiency, communication efficiency, individual rationality and distribution of computation. The other important parameter on which negotiation could take place include,

- Cardinality of negotiation (1 to 1, 1 to many, and many to many)
- Agent characteristics (role, rationality, knowledge, commitment)
- Environment and goods characteristics (private public value, numbers)
- Event parameters (importance of time, schedules)
- Information parameters (price quotes, transaction history (experience))

The involvement of many parameters makes automated negotiation a really complex process. The broader issues to be addressed include, development of good negotiation techniques, which would suit the specific domain, relative strengths and weaknesses of each technique should be thoroughly explored and generation of predictable negotiation

behaviours need to be developed. "At present there has been virtually no work on how a user can instruct an agent to negotiate on their behalf" (Jennings GDN2000).

There have been different approaches proposed for integrating these intelligence factors like negotiation strategies, trade-off mechanisms [28] [29], different negotiating functions [1] and tactics in these agents. *Genetic algorithms* are one of the approach suggested for negotiating aspects where these agents learn over the time by crossovers and mutations there by improving their negotiating capabilities [9] [29]. *Reinforcement learning* is another approach based on rewarding actions that turn out to be positive and punishing those that are negative [8]. *Rule based learning* is another approach for negotiating agents in a virtual market place [16] which is based on particular rules in the system and it proved to be effective.

Appendix C Case-Based Reasoning

Case-Based Reasoning is fundamentally different from other AI approaches in many respects. Case-Based Reasoning helps in solving a new problem by remembering a previous similar situation and by reusing information and knowledge of that situation. When a new problem is presented, a CBR system solves the problem by finding a similar past case, and reusing it in the new problem situation.

CBR is used most commonly in day-today commonsense reasoning. For example if we decide on receiving certain service from a firm or individual based on our previous experience with them. Its is the common sense that the second time we solve the problem or do the same task much easier than the first time.

There are two primary motivations behind CBR, one is Cognitive Science, the desire to model human reasoning behaviour and the second is Artificial Intelligence. The CBR paradigm has been used to perform a variety of problem solving tasks like planning, design and diagnosis. CBR can advance AI technology in the following five areas [14]:

1. Knowledge acquisition
2. Knowledge maintenance
3. Increasing problem-solving efficiency
4. Increasing quality of solutions
5. User acceptance

CBR gets its theoretical support from the *Theory of Dynamic memory* proposed by R. Schank et al. The main premise of the theory is that remembering, understanding, experiencing and learning cannot be separated from each other. The theory suggests that, our memories, which are dynamic, change as a result of our experiences due to the things we encountered, the question that arise on out minds as a result of experiences, and the way we answer these questions. We understand by trying to integrate new things we encounter with what we already know. Thus, understanding causes us to remember old experiences, consciously or unconsciously, as we process the new ones. Understanding, in turn, allows memory to recognize and refine itself, in other words, to be dynamic.

C.1 Case Representation

A case represents specific knowledge tied to a context. It records knowledge about how a task was carried out or how a piece of knowledge is applied or what particular strategies for accomplishing a goal were used. It has to be noted that not every situation is worthy of recording as a case. A stored case must teach a useful lesson in one way or another. A case i.e. an 'experience' may contain three major components.

- A situation or problem description
- A solution
- An outcome

C.2 Case Indexing

An important issue in case-based reasoning is retrieval of appropriate cases from the case memory. The cases have to be indexed to retrieve appropriate cases at situations. Some of the guidelines to index cases are,

- Index should be predictive
- Indexes should be abstract enough to make a case useful in a variety of future situations.
- Indexes should be concrete enough to be easily recognizable in future situations.

C.3 Case Retrieval

One of the most important components of the case-retrieval process is comparing individual cases stored in the case library against the new situation and evaluating the degree of match that is done by matching and ranking procedures.

C.4 Case Adaptation

In CBR old solutions are used as inspiration for solving new problems. Old situations must be modified to fit the new situations, which is called Adaptation. Adaptation plays a fundamental role in the flexibility of problem solving CBR system, and constitutes the major pat of Case Reuse. The major steps in Adaptation include, figuring out what needs to be adapted, and what kind of action needs to be taken.

C.5 Case Evaluation

This phase suggests a solution for the new problem. This phase gives the reasoner a way to evaluate its decisions in the real world, allowing it to collect feedback that enables it to learn. Evaluation is process of judging the goodness of proposed solution.

Evaluation includes explaining differences, justifying differences, projecting outcomes, and comparing and ranking alternative possibilities.

C.6 Case Retainment

This phase constitutes the learning ability of the system. The learning from success or failure of the proposed solution is triggered by the outcome of the evaluation and possible repair. It involves selecting which information from the case to retain, in what form to retain it, how to index the case for later retrieval for similar problems, and how to integrate the new case in the memory structure.

There are many case-based reasoning systems successfully existing and some of them have been put to commercial use. They include [19]: (1) Clavier, a shop floor assistant, (2) SMART, an integrated call-tracking system and problem solving system, (3) Prism, classifies route bank telexes, (4) CAROL intended to use class descriptions in object oriented programming. Other applications where case based reasoning has been used include Photocopiers, Aircraft maintenance, and Medical diagnosis.

From the above discussion we can infer that "case-based reasoning is both cognitively plausible model of reasoning and a method for building intelligent systems. It is grounded in commonsense premises and observations of human cognition and has applicability to a variety of reasoning tasks, providing for each a means of attaining increased efficiency an better performance [14]." Case-based reasoning integrates problem solving, understanding, learning, and memory in to one framework.

Appendix D Results

In testing the proposed framework, following are the samples of experiments conducted. We populated the database with the information about the products and conceptual similarity to represent the ontology. Number of cases are populated in buyer and seller experience databases. To test the functionality of this framework, we assumed certain real time situations and created the agents with the assumed data. We supplied the agents with both successful and unsuccessful experiences. We simulated different situations, where one-one, one-many, many-many negotiations are represented.

D.1 Experiment using an Unsuccessful Experience to buy Similar Product

Assume a situation where a user wants to buy a monitor and the attitude of the user is ⟨0.6, 0.8,0.6⟩. Following are the set of cases present in the Casebase for a buyer.

Pcat	Pname	War	PPR	Price	Att	Sdr	NegRec	Res
Compprods	Printer	2	300-450	400	.9,.5,.9	1	320/360/400	1
Comprods	Mouse	.5	8-15	13	.9,.2.9	1	10/13	1
Homeappl	Fridge	2	400-550	520	.9,.1,.9	.33	415/455/490/ 520	1
Homeappl	Fridge	2	400-550	435	.1,.9.4	.33	405/415/425/ 435	0
Homeelec	TV	1	300-500	420	.7,.5,.7	1.5	310/345/380/ 420	1
Compprod	Printer	1	300-450	335	.2,.8,.3	.5	305/320/335	0

Table D.1 Buyer Experiences in database.

In a similar situation a seller would like to sell a monitor with an attitude ⟨0.2, 0.9, 0.4⟩ and the following are the cases in the case base.

Pcat	Pname	War	PPR	Price	Att	Sdr	NegRec	Res
Compprods	Printer	1.5	300-450	360	.8,.3,.9	2	420/390/360	1
Comprods	Mouse	1	8-15	10	.4,1,.8	3	15/13.5/11/10	0
Homeappl	Fridge	2	400-550	515	.2,.9,.5	2	545/535/525/ 515	0
Homeappl	Fridge	3	400-550	465	.8,.6.8	1	540/515/490/ 465	1
Homeelec	TV	2	300-500	405	.5,.5,.5	1	495/475/455/ 435/405	1
Compprod	Printer	1	300-450	420	.1,1.0,.3	2	450/440/430/ 420	0

Table D.2 Seller Experiences in Database

Assume that the current SupplyDemandRatio is 1 and current Warranty is 1 year. Then using the following steps agents calculate their price ranges.

1. First they retrieve the public price ranges for monitors from the ontology (Refer to figure 4.6 in page.#. 52).
2. Then they retrieve list of similar products form the ontology. In this case Printer and Monitor and retrieved as similar products from the ontology.
3. Now they search the Casebase for best experience. As there is no experience of buying a monitor in the Casebase, the next similar match is buying a printer. This case is matched with 0.78 similarity to the current situation in case of the buyer.

4. Then the buyer agent uses this experience and adjusts its attitude slightly as discussed in 3.6.2.1 of chapter 3.
5. With the new adjusted attitude the agents calculate their APRmin and APRmax using APR function discussed in 3.4.6 of chapter 3
6. Then the agent calculates its bid increment for each proposal and counter proposal as described in 3.6.2.3 of chapter 3.
7. Then buyer agent and seller agent exchange offers and counteroffers till a deal is reached.

Following are the steps involved for buyer and seller in finishing the deal assuming that it is one to one negotiation (no other seller and buyers in the market for the same product)

	Buyer	**Seller**
Current attitude:	t = 0.6, p = 0.8, c = 0.6	t = 0.2, p = 0.9,c = 0.4
Product:	Monitor	Monitor
Exp Picked:	(the last record in buyer experience table above in figure D.1)	(the last record in seller experience table above in figure D.2)
APR:	Min – 256, Max – 317.	Min – 292, Max – 348.
Bid De/Increment:	$16	$12
Negotiation Record:		
	256 ↔	348
	272 ↔	336
	288 ↔	324
	304 ↔	accepted

The negotiation is ended as a success after four exchanges. Note that both seller and buyer don't have the experience of buying a Monitor. Moreover, they both used an unsuccessful experience in adjusting their attitude and in generating the solution. But using an experience of buying a Printer, they adjusted their attitude and calculated their Max, Min and Bid Increments. They ended up in a successful negotiation as buyer's proposal reached seller's minimum. This negotiation confirms that the agents in this framework use the experience of buying or selling similar products. It is interesting to note the way the agents in this framework pick up the relevant experience. More experiments need to be conducted using various matching weights to monitor the behavior of these agents in picking up experiences.

This new experience is stored in the case base by both buyer and seller as new experience as none of the cases in the Casebase matches this negotiation.

Pcat	Pname	War	PPR	Price	Att	Sdr	NegRec	Res
Compprods	Monitor	1	250-350	304	.6,.7,.6	1	256/272/288/ 304	1

Table D.3 A new Experience of Buying a Monitor

Before we continue further, we need to make a point about merging two similar cases, since it is a part of learning process in this framework. Case updates in the Casebase represent the refinement of experiences.

D.2 Merging Cases

The resulted negotiation discussed in previous section is stored in the case base for a buyer as his experience. Assume that the buy would like to buy another monitor, and

his current attitude is ⟨0.9, 0.1, 0.9⟩. With this attitude the buyer tend to be aggressive in bidding. If the buyer finds the a seller with same attitude as the seller in last example in previous section, then this interaction would open up interesting situation. The negotiation would be as follows:

	Buyer		**Seller**
Product:	Monitor		Monitor
Attitude:	t = 0.9, p = 0.1, c = 0.9		t = 0.2, p = 0.9,c = 0.4
Exp picked:	(buyer would use the exp in Figure. D.3)		(assume seller would use the same Experience from last time)
APR:	Min – 257, Max – 347		Min – 292, Max – 348.
Bid De/Increment:	$19		$12
Negotiation:	257	◄────►	348
	276	◄────►	336
	accepted	◄────►	accepted

This negotiation has ended in just 3 exchanges. It is interesting to note that as the buyer agent is more aggressive in bidding when compared to last example, he got the deal for slightly more price in less time. But the main focus here is "how the buyer is going to store his new experience". After matching this new experience with the cases in the Casebase, this experience very strongly matches with the case shown in Figure D.3. The *Merge* function described in section 3.6.3.2 of chapter 3 would merge both cases as follows:

Merge(⟨compprods,monitor,1,250-350,336,(.8,.2,.9),1,257/276/336⟩,
⟨compprods,monitor, 1,250-350, 304,(.6,.7,.6),1, 256/272/288/304)⟩)

$$= \begin{bmatrix} \text{lub}(comprods, comprods), (monitor = monitor), (1,1)(250-350 = 250-350), \\ avg(336,304), avg(.8,.2,.9,,.6,.7,.6), avg(1,1), \min(3,4) \end{bmatrix}$$

The case in the database in Figure D.3 will be updated as follows:

Pcat	Pname	War	PPR	Price	Att	Sdr	NegRec	Res
compprods	Monitor	1	250-350	320	.7,.5,.7	1	257/276/336/	1

Table D.4 Updated Experience of Buying a Monitor

The previous experience in Casebase is updated when there is a strong match between two cases. This experience assumed to be a refined experience and will be used in future negotiations.

D.3 Many-Many interactions

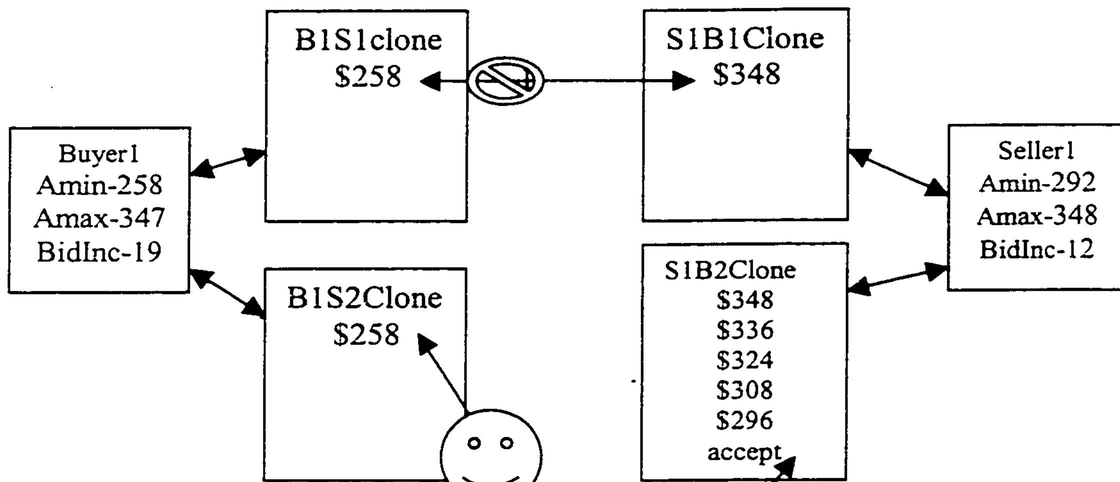
Many to Many interactions might help agents to finish negotiation at a faster rate. We consider a situation where two buyers and two sellers are competing for the same product with different attitudes.

Assume that the same buyer and seller are in the market from the last example where merging of cases is explained. Now we consider one more a buyer and a seller to simulate two-two interactions in buying a monitor. Attitude, APR, and Bid increment for each agent is as follows:

	Buyers		**Sellers**
Buyer1:	Att: t,p,c = (0.9,0.1,0.9)	***Seller1:***	Att: t,p,c = (0.2,0.9,0.4)
	APRMin = 258		APRMin = 292
	APRMax = 347		APRMax = 348
	Bid Increment = $19		Bid Decrement = $12

Buyer2:	Att: t,p,c = (0.2,0.9,0.4)	***Seller2:***	Att: t,p,c = (0.9,0.2,0.8)
	APRMin = 253		APRMin = 255
	APRMax = 294		APRMax = 342
	Bid Increment = $9		Bid Decrement = $29

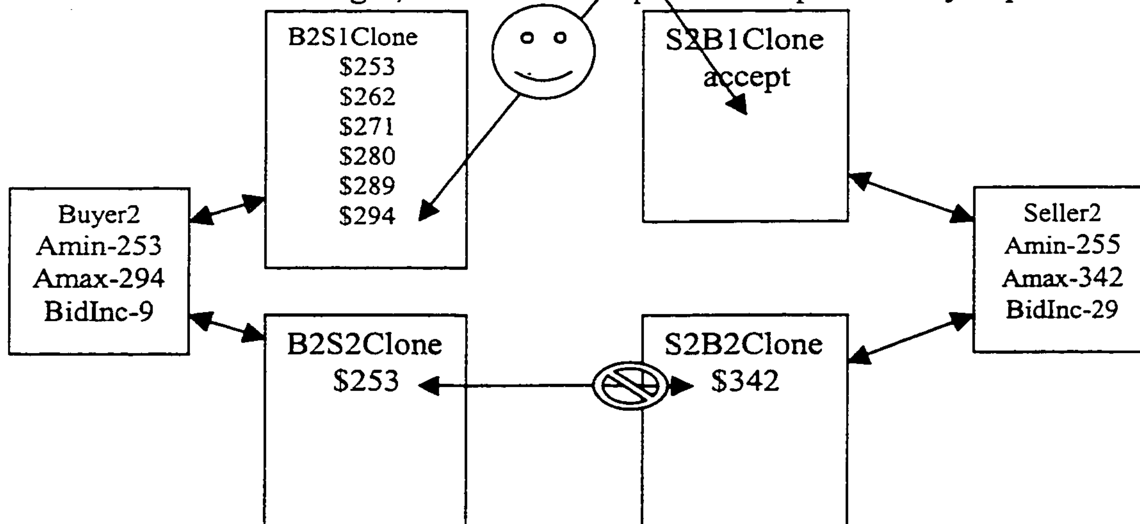
Interaction between the buyers and sellers is as follows in this situation.



Deal done in one exchange, as time is important for both

Agents Clones Clones Agents

Deal done after 6 exchanges, as time is not important and price is very important



The above agent interaction suggests that Buyer1 and Seller2 finished their negotiation in one exchange, as time is very important for them. Note that in the previous example the buyer had to involve in three exchanges as the seller negotiating with him had more priority for price and less priority for time. But in this interaction, the buyer agent finished the deal really quick as it found the seller with an optimal attitude to suit its priorities. Buyer2 and Seller1 finished their negotiation after 6 exchanges. Interestingly, all of them ended up in successful negotiation. Thus many-many interactions might help agents to finish negotiations at faster rate. We tested various scenarios of interactions to assess the behaviour of these agents. As there are no random variables in this model, experimental results were in line with the predictions.

A point has to be made about judging the goodness of a deal. There is a need for some utility functions, which would evaluate the deal so that we can determine if these agents are making successful deals over time. Currently this framework does not support the evaluation of deal, therefore its is hard to predict if there agents are actually getting better over time. It would be interesting to incorporate few objective functions and evaluate the deal the agents make, so that we can test the real learning in these agents.

Appendix E Documented Code

All the important classes in this implementation are shown below. Code for all-important interfaces created in this prototype are given first and then their implementation is given.

```
********************************
Interface Imediator
*********************************
package mediator;

import java.util.*;
import buyer.IBuyer;
import seller.ISeller;
import java.rmi.*;

public interface IMediator extends Remote
{
public String setBuyer(String product,String serverName,String name)throws
RemoteException;
public String setSeller(String product,String serverName,String name,String warr)throws
RemoteException;
   public Vector getCount(String productName) throws RemoteException;
   public void removeBuyer(String id)throws RemoteException;
   public void removeSeller(String product,String id,boolean flag)throws
   RemoteException;
}

********************
Interface IBuyer

********************
package buyer;

import java.util.*;
import seller.ISeller;
import java.rmi.*;
import product.ProductInfo;

public interface IBuyer extends Remote
{
public void setSeller(String seller,String warr,Vector count)throws RemoteException;
   public ProductInfo getProduct()throws RemoteException;
   public void acceptProduct(String dispalyValue) throws RemoteException;
   public void rejectProduct(String dispalyValue) throws RemoteException;
   public void soldProduct(String dispalyValue) throws RemoteException;
   public boolean isAccepted() throws RemoteException;
   public void upDateDataBase(boolean negFlag, String negRecord) throws
   RemoteException;
```

```
}

********************
Interface ISeller
********************

package seller;

import java.util.*;
import java.rmi.*;
import product.*;

public interface ISeller extends Remote
{
   public ProductInfo getProduct()throws RemoteException;
   public String setBuyerBroker(String bKey,double sDR) throws RemoteException;
   public void acceptProduct(String dispalyValue) throws RemoteException;
   public void rejectProduct(String dispalyValue) throws RemoteException;
   public void soldProduct(String dispalyValue) throws RemoteException;
   public boolean isAccepted() throws RemoteException;
   public void upDateDataBase(boolean negFlag, String negRecord) throws
   RemoteException;
}

********************
Interface IBuyerBroker
********************

package broker;

import java.util.*;
import seller.ISeller;
import java.rmi.*;

public interface IBuyerBroker extends Remote
{

  public void setSeller(String sellerBrokerKey) throws RemoteException;
  public boolean acceptProduct() throws RemoteException;
  public void reject() throws RemoteException;
  public void setSellerPrice(double maxPrice) throws RemoteException;
  public void sold() throws RemoteException;

}

********************
Interface ISellerBroker
********************

package broker;
```

```
import java.util.*;
import seller.ISeller;
import java.rmi.*;

public interface ISellerBroker extends Remote
{
   // public void setBuyer() throws RemoteException;
   public boolean acceptProduct() throws RemoteException;
   public void rejectProduct() throws RemoteException;
   public void setBuyerPrice(double maxPrice) throws RemoteException;
   public void sold() throws RemoteException;

}
```

****

Class ApplicationMediator (implements the functionality of Environment discussed in this framework

****

```
package mediator;

/**
This class is the Environment, referred in the framework. Buyers and Sellers register and
informs the buyer about the sellers in the market selling the same product.
*/


import java.util.*;
import buyer.IBuyer;
import seller.ISeller;

import java.rmi.*;
import java.rmi.server.*;
import java.io.*;
import java.net.*;


public class ApplicationMediator extends UnicastRemoteObject implements IMediator
{

   private Map listOfBuyers = null;
   private Map listOfSellers = null;
   private Vector removeBuyer = new Vector();
   private Vector removeSeller = new Vector();
   private static int countSellerConnection = 10000;
   private static int countBuyerConnection = 1000;
   public int sleepTime = 500;
   public static String ip = "//localhost/mediatorServer";
   public boolean isActive = false;
   public boolean isLookup = false;
   public ApplicationMediator()throws RemoteException
   {
```

```
    super();
    (new SellerLookUp(this)).start();
}
/**
This method is invoked by any seller to register their product
*/
 public  String setSeller(String product,String serverName,String name,String
 warr)throws RemoteException

{
   while(isLookup)
   {
      try{
        Thread.sleep(10);
              }catch(InterruptedException iEx){iEx.printStackTrace();}
   }
   isLookup = true;
   isActive = true;
   countSellerConnection = countSellerConnection +1;
   ProductServer prodServ = new ProductServer(product,
   serverName+""+countSellerConnection,name,""+countSellerConnection,warr);
   putSeller(""+countSellerConnection , prodServ );
   System.out.println("in setSeller");
   isActive = false;
   isLookup = false;
   return ""+countSellerConnection;

}
/**
This method is invoked by any Buyer to register
*/
public  String setBuyer(String product,String serverName,String name)throws
RemoteException

{
   while(isLookup)
   {
      try{
        Thread.sleep(10);
              }catch(InterruptedException iEx){iEx.printStackTrace();}
   }
   isLookup = true;
   isActive = true;
   countBuyerConnection = countBuyerConnection +1;
   ProductServer prodServ = new ProductServer(product,
   serverName+""+countBuyerConnection,name,""+countBuyerConnection);
   putBuyer(""+countBuyerConnection , prodServ );
   System.out.println("in setBuyer");
   isActive = false;
   isLookup = false;
   return ""+countBuyerConnection;
}

/**
```

```
This method informs the environment that the buyer is exiting the market
*/

public synchronized void removeBuyer(String id)throws RemoteException
{
   while(isLookup)
   {
      try{
         Thread.sleep(10);
               }catch(InterruptedException iEx){iEx.printStackTrace();}
   }
   isActive = true;
   isLookup = true;
   removeBuyer.add(id);
   System.out.println("in removeBuyer");
   isActive = false;
   isLookup = false;
}
/**
This method informs the environment that the seller is exiting the market
*/

public synchronized void removeSeller(String inProduct,String id,boolean
flag)throws RemoteException

{
   while(isLookup)
   {
      try{
         Thread.sleep(10);
               }catch(InterruptedException iEx){iEx.printStackTrace();}
   }
   isActive = true;
   isLookup = true;
   removeSeller.add(inProduct+";"+id+";"+flag );
   isActive = false;
   isLookup = false;
}
/**
This method returns the number of sellers and number of buyers for same product
*/
public synchronized Vector getCount(String productName)throws RemoteException
{
  int sellerCount = 0;
  int buyerCount = 0;
  Iterator itS = listOfSellers.keySet().iterator();
  while(itS.hasNext())
   {
      String keyS = (String)itS.next();
      ProductServer sellValue = getSeller(keyS);
      if(productName.equalsIgnoreCase(sellValue.getproduct(0)))
         sellerCount = sellerCount +1;
   }
   Iterator itB = listOfBuyers.keySet().iterator();
```

```
      while(itB.hasNext())
      {
         String keyB = (String)itB.next();
         ProductServer buyValue = getBuyer(keyB);
          if(productName.equalsIgnoreCase(buyValue.getproduct(0)))
            buyerCount = buyerCount +1;
      }
      Vector data = new Vector();
      data.add(""+sellerCount);
      data.add(""+buyerCount);
      return data;
   }
   /**
   This method is invoked by environment to whenever a seller or buyer register in the
environment
   */
   public void isSellerFound()
   {
     upDateBuyer();
     upDateSeller();
     if(listOfSellers == null || listOfBuyers == null)
     {
        return;
     }
     Iterator itS = listOfSellers.keySet().iterator();
     boolean flag = true;
     while((flag && itS.hasNext()))
      {
        String keyS = (String)itS.next();
        ProductServer sellValue = getSeller(keyS);
        Iterator itB = listOfBuyers.keySet().iterator();
        while((flag && itB.hasNext()))
           {
              String keyB = (String)itB.next();
              ProductServer buyValue = getBuyer(keyB);
              if(sellValue.isEqual(buyValue.getproduct(0),buyValue.getId()))
              {
                 System.out.println("Found same product for seller = "+sellValue.getId()
                  +"and the buyer = "+buyValue.getId());
        buyValue.setListServerId(sellValue.getId());
                 sellValue.setListServerId(buyValue.getId());
                    (new RunBuyer(sellValue,buyValue)).start();
                 //flag = false;
              }
           }
         }

   }
   /**
   This method informs the buyer about a relevant seller registered in the environment
   */

   public synchronized void makeConnection(ProductServer seller ,ProductServer
   buyer)
```

```
{
  try {
      // lookup  buyer.getServerName remote object in rmiregistry
      IBuyer myBuyer = (IBuyer)Naming.lookup( buyer.getServerName());

myBuyer.setSeller(seller.getServerName(),seller.getWarrant(),getCount(seller.getproduct(0)));
    }catch ( java.rmi.ConnectException ce ) {
     System.err.println( "Connection to server failed. " +
       "Server may be temporarily unavailable." );
    }catch ( Exception e ) {
     e.printStackTrace();
     System.exit( 1 );
     }
}
private void upDateBuyer()
{
   if(removeBuyer.size() <= 0)
      return;
   int size = removeBuyer.size();
   for( int j =0 ; j <size; j++)
   {
      String id = (String)removeBuyer.get(j);
      ProductServer prodServ = (ProductServer)getBuyer(id);
      String product = prodServ.getproduct(0);
      Vector vList = prodServ.getListOfServerId();
      int vSize = vList.size();
      for(int i=0 ; i< vSize; i++)
      {
         removeBuyerFromSellerList(getSeller((String)vList.get(i)),product,id);
      }
      removeBuyerFromList(id);
   }
   removeBuyer = new Vector();
   System.out.println("in upDateBuyer");
}

private void upDateSeller()
{
   if(removeSeller.size() <= 0)
      return;
   int size = removeSeller.size();
   for( int j =0 ; j <size; j++)
   {

      String value = (String)removeSeller.get(j);
      StringTokenizer token = new StringTokenizer(value,";");

      String id =  (String)token.nextElement();
      String inProduct = (String)token.nextElement();
      boolean flag = Boolean.getBoolean((String) token.nextElement());
      Iterator itB = listOfBuyers.keySet().iterator();
      while(itB.hasNext())
      {
```

```
            String keyB = (String)itB.next();
            ProductServer buyValue = getBuyer(keyB);
            Vector vId = buyValue.getListOfServerId();
            String bId = (String)vId.get(0);
            if( bId.equalsIgnoreCase(id))
            {
               removeSellerFromBuyerList(buyValue,inProduct,id);
               if(!flag)
               {
                  removeBuyerFromSellerList(getSeller(id),inProduct,id);
                  System.out.println("in removeSellerFromBuyerList");
               }
            }
         }
         if(flag)
         {
            removeSellerFromList(id);
         }
      }
      removeSeller = new Vector();
      System.out.println("in upDateSeller");
   }
   private void removeBuyerFromSellerList(ProductServer seller ,String rmProduct,String id)
   {
      seller.removeListOfServerId(id);
      System.out.println("in removeBuyerFromSellerList");
   }
   private void removeSellerFromBuyerList(ProductServer buyer ,String rmProduct,String id)
   {
      buyer.removeListOfServerId(id);
      System.out.println("in removeSellerFromBuyerList");
   }

   public synchronized void putSeller(String key, ProductServer value)
   {
      if(listOfSellers == null)
         listOfSellers = new HashMap();
      listOfSellers.put(key,value);

   }
   public synchronized void putBuyer(String key,  ProductServer value)
   {
      if(listOfBuyers == null)
         listOfBuyers = new HashMap();
      listOfBuyers.put(key,value);
   }
   public synchronized void removeSellerFromList(String id)
   {
      if(listOfSellers == null)
        return;
      listOfSellers.remove(id);

   }
   public synchronized void removeBuyerFromList(String id)
```

```
{
   if(listOfBuyers == null)
      return;
   listOfBuyers.remove(id);
}

public synchronized ProductServer getSeller(String key)
{
   if(listOfSellers != null)
      return (ProductServer)listOfSellers.get(key);
   return null;
}
public synchronized ProductServer getBuyer(String key)
{
   if(listOfBuyers != null)
      return (ProductServer)listOfBuyers.get(key);
   return null;
}

static public void main(String args[])throws Exception
      {
            try
            {
         System.err.println("Initializing server: please wait." );
   // create server object
   ApplicationMediator mediator = new ApplicationMediator();
   // bind IMediator object to the rmiregistry
   String serverObjectName = ip ;
   Naming.rebind( serverObjectName, mediator );
   System.err.println("The ApplicationMediator Server is up and running." );
            }
            catch (Throwable t)
            {
                  System.err.println(t);
                  t.printStackTrace();
                  //Ensure the application exits with an error condition.
                  System.exit(1);
            }
      }

      class ProductServer
      {
         private Vector product = new Vector();
         private String name = null;
         private String serverName = null;
         private String id = "";
         private String warrant = "0";
         private Vector listServerId = new Vector();

         ProductServer(String inProduct,String inServerName,String inName,String
         inId)

         {
            this(inProduct,inServerName,inName,inId,"0");
```

```
}
ProductServer(String inProduct,String inServerName,String inName,String
inId,String warr)

{
   product.add(inProduct);
   serverName = inServerName;
   name = inName;
   id = inId;
   warrant = warr;
}
public void removeListOfServerId(String id)
{
   listServerId.remove(id);
}
public void setListServerId(String id)
{
   listServerId.add(id);
}
public void setProduct(String inProduct)
{
   product.add(inProduct);
}
public Vector getListOfServerId()
{
   return listServerId;
}
public String getServerName()
{
   return serverName;
}
public String getproduct(int i)
{
   return (String)product.get(i);
}
public String getName()
{
   return name;
}
public String getId()
{
   return id;
}
public String getWarrant()
{
   return warrant;
}
public boolean isProductFound(String inProd)
{
   return product.contains(inProd);
}
public boolean isEqual(String inProduct,String inId)
{
   boolean match = product.contains(inProduct);
```

```
            if( match)
            {
                if(isServerIdFound(inId))
                {
                    match = false;
                }
            }
            return match;
        }
        private boolean isServerIdFound(String inId)
        {
            int size = listServerId.size();
            for( int i = 0; i < size; i++)
            {
                String sId = (String)listServerId.get(i);
                if(sId.equalsIgnoreCase(inId))
                    return true;
            }
            return false;
        }

    }
    class SellerLookUp extends Thread
    {
        ApplicationMediator med = null;
        SellerLookUp(ApplicationMediator med)
        {
            super();
            this.med = med;
        }
        public void run()
        {
            while(true)
            {
                while(isActive)
                {
                    try{
                        Thread.sleep(sleepTime);
                    }catch(InterruptedException iEx){iEx.printStackTrace();}
                }
                try
                {
                    isLookup = true;
                    if(med != null)
                    {
                        med.isSellerFound();
                    }
                    isLookup = false;
                    Thread.sleep(sleepTime);
                }catch(InterruptedException iEx){iEx.printStackTrace();}

            }
        }
    }
```

```
    class RunBuyer extends Thread
    {
       ProductServer buy = null;
       ProductServer seller = null;
       RunBuyer(ProductServer seller,ProductServer buy)
       {
          super();
          this.buy = buy;
          this.seller = seller;
       }
       public void run()
       {
   // try{
      makeConnection(seller,buy);
   //}catch(RemoteException ee){ee.printStackTrace();}

       }
    }


}
```

****

Class Ontology implements the functionality of Ontology discussed in the framework to supply domain specific knowledge

****

```
package util;
/**
This class is used to model the ontology where similar products, and
information about the products is returned from the database, using JDBC.
*/
import java.util.*;
import java.sql.*;
import java.net.URL;
import java.io.*;

public class Ontology
{
   /**
   This method returns a vector of similar products
   */
  public Vector RetSimProducts(String pname)
  {

          Connection conn;
          String url = "jdbc:odbc:orawin95";
          String uname = "scott";
          String upwd = "tiger";
          Vector Similarproducts = new Vector(10);
          try
```

```
{
   try
   {
          Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
     }

     catch (ClassNotFoundException cnfex){
          System.err.println("Failed to load JDBC/ODBC driver.");
           cnfex.printStackTrace();
                          }

      conn = DriverManager.getConnection(url,uname,upwd);
      System.err.println("Connected ....");
      Statement stmt = conn.createStatement();

      ResultSet rs = stmt.executeQuery("Select * from range where
                    prodname = '"+pname+"' ");

      double Pmaxactu=0;
      double Pminactu=0;

      String Productcat1 ="";
      while (rs.next())
      {
        Productcat1 = rs.getString("Productcat");
        Pmaxactu = rs.getFloat("minprice");
        Pminactu = rs.getFloat("maxprice");
      }
      double Averpriceactu;

      Averpriceactu =(double)( Pmaxactu + Pminactu )/(double)2.0;

ResultSet rsim = stmt.executeQuery("Select Prodcat2, value from
simvalue where prodcat1 = '"+Productcat1+"' and value >= 0.9" );

      Vector Simcats = new Vector (10);

      Concepsim consim = new Concepsim();
      while (rsim.next())
      {

        consim.prodcat1 = Productcat1;
        consim.prodcat2 = rsim.getString("Prodcat2");
        consim.simvalue = rsim.getInt("value");
        Simcats.addElement(consim);
      }

      Enumeration simcat = Simcats.elements();

      Concepsim prosim;


      while (simcat.hasMoreElements())
      {
```

```
                prosim = (Concepsim)simcat.nextElement();
                String Prodcat1;
                Prodcat1 = prosim.prodcat1;

                ResultSet simpro = stmt.executeQuery("Select prodname, maxprice,
                minprice from range where productcat = '"+Prodcat1+"' ");
                double Match;

                while (simpro.next())
                {
                   String Prodnameother;
                   Prodnameother =simpro.getString("prodname");
                   double  Pmaxother;
                   Pmaxother = simpro.getFloat("maxprice");
                   double  Pminother;
                   Pminother =simpro.getFloat("minprice");
                   double Avgpriceother ;

                   Avgpriceother = (Pmaxother + Pminother)/(double)2.0;

            Match = productSimilarity(prosim.simvalue,
        Averpriceactu,Avgpriceother);
                   double threshold = (double)0.75;
                   if (Match >= threshold)
                   {
                   Simproduct similarpro = new Simproduct();

                   similarpro.Pname = Prodnameother;
                   similarpro.Pcat = Prodcat1;
                   similarpro.Pmax = Pmaxother;
                   similarpro.Pmin = Pminother;
                   similarpro.PSim = Match;

                   Similarproducts.addElement(similarpro);

                   }


                }

              }

               conn.close();

         } catch (SQLException sqlex)
         {
              System.err.println("Failed to load JDBC/ODBC driver.");
              sqlex.printStackTrace();
       }

   return  Similarproducts;
```

```
}

/**
The product similarity function discussed in chapter3 of the thesis
*/
public double productSimilarity(double value, double price1,double price2)
{
    double prodsimilairty;

    prodsimilairty = value * (1- Math.abs(price1-price2)/Math.max(price1,price2));


    return prodsimilairty;

}
/**
This functions returns the Public price ranges for a given product in the
ontology
*/
public Product getProdDet(String Pname)
{
    Product produ = new Product();
    Connection conn;
    String url = "jdbc:odbc:orawin95";
    String uname = "scott";
    String upwd = "tiger";

        try
        {
            try
          {
                Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            }

            catch (ClassNotFoundException cnfex){
                System.err.println("Failed to load JDBC/ODBC driver.");
                cnfex.printStackTrace();
                            }

            conn = DriverManager.getConnection(url,uname,upwd);
            System.err.println("Connected ....");
            Statement stmt = conn.createStatement();

            ResultSet rs = stmt.executeQuery("Select * from range where
      prodname = '"+Pname+"'");
            String Pmaxactu;
            String Pminactu;
            String Productcat1;
            while (rs.next())
            {
              produ.Pcat = rs.getString("productcat");
              produ.Pname = rs.getString("prodname");

              produ.Pmin = rs.getFloat("minprice");
```

```
                produ.Pmax = rs.getFloat("maxprice");

              }
              conn.close();

          } catch (SQLException sqlex)
                {
              System.err.println("Failed to load JDBC/ODBC driver.");
              sqlex.printStackTrace();
            }
      return produ;

  }

}
```

****

Class Case implements the functionality to retrieve similar experience and updates the experience

****

```
package util;
/**
This class returns best similar case from the case base for a given
situation.  This class uses ontology to get the similar products so taht it
can return similar relevant experience.
*/
import java.sql.*;
import java.net.URL;
import java.io.*;
import java.util.*;

public class Case

{
        Ontology onto;
        Product prod;


        double ImpTimePast;
        double ImpPricePast;
        double CommitPast;
        double PricePast;
        double PricePresneg;
        double PricePres;
        double Pmaxactu;
        double Pminactu;
        double Pmaxother;
        double Pminother;
        double WarrPast;
        double WarrPres;
```

```
        double SdrPast;
        double ProductSim;
        double RangeSim;
        double AttitudeSim;
        double NegSim;
        double Match;
        double MaxMatch =(double) 0.7;

        String Pname;
        String Pcat;
        String NegRec;
        String Bestcase;
        Vector SimProducts;


        public Case()
        {
                onto = new Ontology();
        }
        public void getProductDetails(String Pname)
        {  String tester;
                prod = onto.getProdDet( Pname);
                Pcat = prod.Pcat;
                Pmaxactu = prod.Pmax;
                Pminactu = prod.Pmin;
                tester = prod.Pname;
        }
    public double getPmaxactu()
    {
       return Pmaxactu;
    }
    public double getPminactu()
    {
       return  Pminactu;
    }

    /**
    This method returns the best case from the case base
    */
        public String getBestCase(String User, String Pname,double ImpTimePres,
                double ImpPricePres, double CommitPres )
        {
          SimProducts = onto.RetSimProducts(Pname);
          Enumeration enum = SimProducts.elements();
          Connection conn;
         String url = "jdbc:odbc:orawin95";
         String uname = "scott";
         String upwd = "tiger";

          try
         {
                  try
                  {
                  Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

```
          }

      catch (ClassNotFoundException cnfex){
            System.err.println("Failed to load class driver.");
             cnfex.printStackTrace();
                              }
              conn = DriverManager.getConnection(url,uname,upwd);
       System.err.println("Connected ....");
       Statement stmt = conn.createStatement();


                  while(enum.hasMoreElements())
                  {
Simproduct similarpro;
similarpro = (Simproduct) enum.nextElement();
String PnameOther;
PnameOther = similarpro.Pname;
 Pmaxother = similarpro.Pmax;
 Pminother = similarpro.Pmin;
 ProductSim = similarpro.PSim;
                          //database connectivity string

String statement  = "Select * from sellerexp where PRODNAME ='"+PnameOther+"'";

 if(User.equalsIgnoreCase("buyer"))

statement = "Select * from buyerexp where PRODNAME ='"+PnameOther+"'";


ResultSet rs = stmt.executeQuery(statement );
String Attitudeother;
String Pastneg;
String othercase;
double Pricepast;
String Pcatother;
int PastRes;
double SdrPast;
double WarrPast;


       while(rs.next())
       {

          Pcatother = rs.getString("PRODCAT");
          PnameOther = rs.getString("PRODNAME");
          WarrPast = rs.getFloat("WARRANTY");
          Pmaxother = rs.getFloat("PPRMAX");
          Pminother = rs.getFloat("PPRMIN");
          Pricepast = rs.getFloat("PRICE");
          Attitudeother = rs.getString("ATTITUDE");
          SdrPast = rs.getFloat("SDR");
          Pastneg = rs.getString("NEGREC");
          PastRes = rs.getInt("RESULT");
```

```
                StringTokenizer tokens = new StringTokenizer(Attitudeother,",");

                while(tokens.hasMoreTokens())
                {
                   ImpTimePast= Double.parseDouble(tokens.nextToken());
                   ImpPricePast= Double.parseDouble(tokens.nextToken());
                   CommitPast = Double.parseDouble(tokens.nextToken());
                }
                othercase = (Pcatother + "|" + PnameOther + "|" + WarrPast + "|" +
                Pmaxother+ "|" +  Pminother  + "|" +  Pricepast + "|" + Attitudeother+ "|" +
                SdrPast + "|" +Pastneg + "|" + PastRes );


                AttitudeSim = attitudeSimilarity(ImpTimePast,ImpTimePres,
                              ImpPricePast,ImpPricePres,CommitPast,CommitPres);

                RangeSim = rangeSimilarity(Pmaxactu,Pminactu,Pmaxother,Pminother);


                Match = ((AttitudeSim + RangeSim + ProductSim)/(double)3.0);

                if( Match >= MaxMatch)
                {
                 Bestcase = othercase;
                 MaxMatch = Match;
                }


              }

           }

        conn.close():

      }
                    catch (SQLException sqlex)
     {
            System.err.println("Failed to load JDBC/ODBC driver!!!!!.");
            sqlex.printStackTrace();
      }
 return Bestcase;

 }
      /**
       This method updates the casebase as discussed in merge function in chapter 3
       */
      public void UpdateDatabase(String product,double SdrPres,double WarrPres,String
      negRecord, double impCurrTime,double impCurrPrice,double commit,String User, int
      res)

      {
       SimProducts = onto.RetSimProducts(product);
       Enumeration enum = SimProducts.elements();
```

```
Connection conn;
MaxMatch = 8.5;
String url = "jdbc:odbc:orawin95";
String uname = "scott";
String upwd = "tiger";
String Attitudeother;
String Pastneg;
String othercase;
double Pricepast;
String Pcatother;
String PnameOther;
int PastRes;
double SdrPast;
double WarrPast;
double negsimilarity;
String statement ;
  try
  {
            try
            {
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
      }

      catch (ClassNotFoundException cnfex){
            System.err.println("Failed to load class driver.");
            cnfex.printStackTrace();
                              }
            conn = DriverManager.getConnection(url,uname,upwd);
      System.err.println("Connected ....");
      Statement stmt = conn.createStatement();


                  while(enum.hasMoreElements())
                  {
Simproduct similarpro;
similarpro = (Simproduct) enum.nextElement();
PnameOther = similarpro.Pname;
 Pmaxother = similarpro.Pmax;
 Pminother = similarpro.Pmin;
 ProductSim = similarpro.PSim;
//database connectivity string

statement  = "Select * from sellerexp where PRODNAME ='"+PnameOther+"'";

if(User.equalsIgnoreCase("buyer"))

statement = "Select * from buyerexp where PRODNAME ='"+PnameOther+"'";


ResultSet rs = stmt.executeQuery(statement );


      while(rs.next())
      {
```

```
  Pcatother = rs.getString("PRODCAT");
  PnameOther = rs.getString("PRODNAME");
  WarrPast = rs.getFloat("WARRANTY");
  Pmaxother = rs.getFloat("PPRMAX");
  Pminother = rs.getFloat("PPRMIN");
  Pricepast = rs.getFloat("PRICE");
  Attitudeother = rs.getString("ATTITUDE");
  SdrPast = rs.getFloat("SDR");
  Pastneg = rs.getString("NEGREC");
  PastRes = rs.getInt("RESULT");

  StringTokenizer tokens = new StringTokenizer(Attitudeother,",");

  while(tokens.hasMoreTokens())
  {
     ImpTimePast= Double.parseDouble(tokens.nextToken());
     ImpPricePast= Double.parseDouble(tokens.nextToken());
     CommitPast = Double.parseDouble(tokens.nextToken());
  }
  othercase = (Pcatother + "|" + PnameOther + "|" + WarrPast + "|" +
  Pmaxother + "|" + Pminother  + "|" + Pricepast + "|" +  Attitudeother+  "|" +
 SdrPast+ "|" +Pastneg + "|" + PastRes );

   AttitudeSim = attitudeSimilarity(ImpTimePast,impCurrTime,ImpPricePast,
                         impCurrPrice,CommitPast,commit);
  RangeSim = rangeSimilarity(Pmaxactu,Pminactu,Pmaxother,Pminother);
  negsimilarity = negotiationSimilairty(negRecord,Pastneg,PastRes,res);

  Match = ((AttitudeSim + RangeSim + ProductSim +
  negsimilarity)/(double)4.0);

      if( Match >= MaxMatch)
      {
      Bestcase = othercase;
      MaxMatch = Match;
      }


    }

  }

String Atti;
//Functionality of the megre function discussed in chapter 3 starts here.


        if (MaxMatch > 8.5 && (PnameOther.equals(product)))
        {

                StringTokenizer exper = new StringTokenizer(Bestcase,"|");

                Pcatother = exper.nextToken();
                PnameOther =exper.nextToken();
```

```
WarrPast = Double.parseDouble(exper.nextToken());
Pmaxother = Double.parseDouble(exper.nextToken());
Pminother = Double.parseDouble(exper.nextToken());
Pricepast = Double.parseDouble(exper.nextToken());
Attitudeother = exper.nextToken();
SdrPast = Double.parseDouble(exper.nextToken());
Pastneg = exper.nextToken();
PastRes =Integer.parseInt(exper.nextToken());


StringTokenizer tokens = new
StringTokenizer(Attitudeother,",");


ImpTimePast= Double.parseDouble(tokens.nextToken());
ImpPricePast= Double.parseDouble(tokens.nextToken());
CommitPast = Double.parseDouble(tokens.nextToken());


impCurrTime = (impCurrTime + ImpTimePast)/((double)2.0);
impCurrPrice = (impCurrPrice + ImpPricePast)/((double)2.0);
commit = (commit + CommitPast)/((double)2.0);

String imCurrTime = Double.toString(impCurrTime);
String imCurrPrice = Double.toString(impCurrPrice);
String comit = Double.toString(commit);


Atti =  imCurrTime +imCurrPrice + comit;
WarrPres = (WarrPres + WarrPast)/((double)2.0);
PricePres = (PricePres + Pricepast)/((double)2.0);
SdrPres = (SdrPres + SdrPast)/((double)2.0);
res = Math.min(res, PastRes);

int pres, past;
StringTokenizer tok = new StringTokenizer(negRecord, "/");
pres = tokens.countTokens();

 tok = new StringTokenizer(Pastneg, "/");
past = tokens.countTokens();

if (past < pres)
negRecord = Pastneg;

statement = "update sellerexp set warranty = '"+ WarrPres+"',
price = '"+PricePres+"', attitude = '"+Atti+"', sdr ='"+SdrPres+"',
negrec = '"+negRecord+"', result = '"+res+"' where attitude = '
"+Attitudeother+"'and price = '"+Pricepast+"' and negrec = '"+
Pastneg+"'";

if(User.equalsIgnoreCase("buyer"))

statement = "update buyerexp set warranty = '"+ WarrPres+"',
```

```
                    price = '"+PricePres+"', attitude = '"+Atti+"', sdr
                    ='"+SdrPres+"', negrec = '"+negRecord+"', result = '"+res+"'
                    where attitude = '"+Attitudeother+"' and price = '"+Pricepast+"'
                    and negrec = '"+ Pastneg+"'";

                    stmt.executeQuery(statement);


            }

            else

            {

                    String imCurrTime = Double.toString(impCurrTime);
                    String imCurrPrice = Double.toString(impCurrPrice);
                    String comit = Double.toString(commit);


                    Atti = imCurrTime +imCurrPrice + comit;

                    statement  = "insert into sellerexp values ('"+Pcat+"',
                    '"+product+"', '"+WarrPres+"', '"+Pmaxactu+"','"+Pminactu+"',
                    '"+Pricepres+"',  '"+Atti+"','"+SdrPres+"','"+negRecord+"',
                    '"+res+"')";

                    if(User.equalsIgnoreCase("buyer"))
                    statement  = "insert into buyerexp values ('"+Pcat+"',
                    '"+product+"', '"+WarrPres+"', '"+Pmaxactu+"','"+Pminactu+"',
                    '"+Pricepres+"',  '"+Atti+"','"+SdrPres+"','"+negRecord+"',
                    '"+res+"')";


                    stmt.executeQuery(statement );


            }



  conn.close();

}
            catch (SQLException sqlex)
     {
     System.err.println("Failed to load JDBC/ODBC driver!!!!!.");
     sqlex.printStackTrace();
   }

}
```

```
/**
This function returns similarity discussed in chapter 3 of the
thesis
*/

        public double attitudeSimilarity(double ImpTimePast,double ImpTimePres,double
        ImpPricePast,   double ImpPricePres,    double CommitPast,double CommitPres)
        {
        double attisim;
        attisim = ((1 - Math.abs(ImpTimePast- ImpTimePres))+(1-Math.abs(ImpPricePast-
        ImpPricePres))+   (1 - Math.abs(CommitPast-CommitPres)))/3;
        return attisim;
        }
/**
This function returns similarity of two price ranges discussed in chapter 3 of the
thesis
*/
        public double rangeSimilarity(double Pmaxactu, double Pminactu, double Pmaxother,
        double Pminother )
        {
        double ransim;
        ransim = (1-(Math.abs(Pminactu-Pminother)/Math.max(Pminactu,Pminother)))*
         (1-(Math.abs(Pmaxactu-Pmaxother)/Math.max(Pmaxactu,Pmaxother)));
        return ransim;
        }

        /**
This function returns similarity between two negotiations discussed in chapter 3 of the
thesis
*/

        public double negotiationSimilairty(String negrecpre, String negreccur, int respre, int
rescur)
        {

                double negsimilarity;
                StringTokenizer tokens = new StringTokenizer(negrecpre, "/");
                Vector numofferspre = new Vector();
                while(tokens.hasMoreTokens())
                {
                numofferspre.addElement(Double.parseDouble(tokens.nextToken()));

                }

                StringTokenizer curtokens = new StringTokenizer(negreccur, "/");
                Vector numofferscur = new Vector();
                PricePres = Double.parseDouble( numofferscur.lastElement());

                while(curtokens.hasMoreTokens())
                {
                numofferscur.addElement(Double.parseDouble(curtokens.nextToken()));

                }
```

```
            int prenumoffers;
            int curnumoffers;

            prenumoffers = numofferspre.size();
            curnumoffers = numofferscur.size();


            if (respre != rescur)

            negsimilarity = (1 - (((Math.abs(prenumoffers - curnumoffers)/
                    Math.max(prenumoffers,curnumoffers))* 0.5)+ 0.5));

            else
            negsimilarity = (1 - (Math.abs(prenumoffers - curnumoffers)/
             Math.max(prenumoffers,curnumoffers)));


            return negsimilarity;
        }


}
```

****

Class Buyer

****

```
package buyer;
/**
This class registers the buyer in the environment, then waits for the message from the
environment.
Once the relevant seller is found, then experience is retrived and buyer's attitude is adjusted.
Then price ranges are calculated.  Then clones are created based on number of relevant
sellers and
sends message to the clone and waits for the results. Finally it updates the Casebase.
*/



import java.util.*;
import seller.ISeller;
import mediator.IMediator;
import product.*;
import broker.*;

import java.rmi.*;
import java.rmi.server.*;
```

```
import java.io.*;
import java.net.*;

public class Buyer extends UnicastRemoteObject implements IBuyer
{

   private String seller = null;
   private String mediatorKey = null;
   private IMediator myMediator = null;
   public static String ip = "//localhost/";
   public static String hostIp = "//localhost/mediatorServer";
   public static int numberOfBroker = 1000;
   public Vector listOfBroker = new Vector();
   public BuyerGUI buyerGUI = null;
   private ProductInfo product = null;
   private boolean isAccess = false;
   public boolean accept = false;
   private int numberOfClone = 0;
   private Vector negRecord = new Vector();
   private int res = 0;
   public Buyer(BuyerGUI gui)throws RemoteException
   {
      super();
                  buyerGUI = gui;
   }
   public void setMediatorKey(String key)throws RemoteException
   {
      mediatorKey = key;
   }
   /**
   This method is invoked by the environment to inform the buyer about the
   relevant seller in the market

   */
   public synchronized void setSeller(String seller,String warr, Vector count)throws
RemoteException
   {
      System.out.println("in setSeller");
      this.seller = seller;
                  buyerGUI.updateResult(seller);
      numberOfBroker = numberOfBroker + 1;
      product.setCurrWar(Double.parseDouble(warr));
      int tSeller = Integer.parseInt((String)count.get(0));
      int tBuyer = Integer.parseInt((String)count.get(1));
      product.setSupDemCurrRatio((double)tSeller/tBuyer);
                  product.calculateProdInfo();
      BrokerProductInfo bProd = new BrokerProductInfo();
      bProd.setAprmax(product.getAprmax());
      bProd.setAprmin(product.getAprmin());
      bProd.setSupDemPastRatio(product.getSupDemPastRatio());
      bProd.setSupDemCurrRatio(product.getSupDemCurrRatio());
      bProd.setImpCurrTime(product.getImpCurrTime() );
      bProd.setImpPastTime(product.getImpPastTime());
      bProd.setImpCurrPrice(product.getImpCurrPrice());
```

```
        bProd.setCurrWar(product.getCurrWar());
        bProd.setPastWar(product.getPastWar());
        bProd.setAlpha(product.getAlpha());
        bProd.setProduct(product.getProduct());
        BuyerBroker bBroker =  new BuyerBroker(
ip+getMediatorKey(),ip+getMediatorKey()+""+numberOfBroker,getSeller(),hostIp,bProd);
                listOfBroker.add(bBroker);
    }
    public String getMediatorKey()
    {
        return mediatorKey;
    }
    public String getSeller()
    {
        return seller;
    }
    public  ProductInfo getProduct()throws RemoteException
    {

            return  (ProductInfo)product;
    }
    /**
    This method invoked by the clone to inform the buyer about the deal done
    successfully.
    */
    public synchronized void acceptProduct(String dispalyValue) throws RemoteException
    {
        System.out.println("in Buyer.acceptProduct");
        while(isAccess)
        {
            try{
                wait();
            }catch(InterruptedException iEx){iEx.printStackTrace();}
        }
        isAccess = true;

        if( (!accept) )
        {
            accept = true;
            buyerGUI.updateResult("Product is bought",dispalyValue);
            myMediator.removeBuyer(mediatorKey);
        }
        isAccess = false;
        notify();
        System.out.println("in Buyer.acceptProduct accept = "+accept);
    }
    /**
    This method invoked by the clone to inform the buyer about the deal done unsuccesfully.
    */

    public synchronized void rejectProduct(String dispalyValue) throws RemoteException
    {
        while(isAccess)
        {
```

```
            try{
                wait();
            }catch(InterruptedException iEx){iEx.printStackTrace();}
        }
        isAccess = true;
        buyerGUI.updateResult("Rejected the offer by the seller",dispalyValue);
        isAccess = false;
        notify();
    }
    /**
    This method invoked by the clone to inform the buyer to inform him that the product is sold to
    other buyer
    */
    public synchronized void soldProduct(String dispalyValue) throws RemoteException
    {
        while(isAccess)
        {
            try{
                wait();
            }catch(InterruptedException iEx){iEx.printStackTrace();}
        }
        isAccess = true;
        buyerGUI.updateResult("Product is sold to other buyer",dispalyValue);
        isAccess = false;
        notify();

    }
    public synchronized boolean isAccepted() throws RemoteException
    {
        return accept;
    }


        public void makeConnection(ProductInfo pro)
    {
     try {
          // name of remote server object bound to rmi registry
          String serverObjectName = hostIp;
          // lookup hostIp remote object in rmiregistry
          product = pro;
          System.out.println("Looking up server "+hostIp);
          myMediator = (IMediator)Naming.lookup( serverObjectName );
          mediatorKey = myMediator.setBuyer(product.getProduct(),ip,product.getName());
          System.out.println("Found server "+hostIp);
          startServer();
          buyerGUI.setTitle("Buyer ID = "+getMediatorKey());
       }catch ( java.rmi.ConnectException ce ) {
        System.err.println( "Connection to server failed. " +
          "Server may be temporarily unavailable." );
       }catch ( Exception e ) {
        e.printStackTrace();
        System.exit( 1 );
        }
    }
```

```
/**
This method is informed by each clone about their negotiation record and also to result
negotiation and then to update the casebase.
*/


public void upDateDataBase(boolean negFlag, String negRec)
{
   if(negFlag)
   {   res = 1;
      numberOfClone = -1;
      product.writeToDataBase(negRec,res);
   }
   else
   {
      negRecord.addElement(negRec);
      if(numberOfClone != -1 && numberOfClone == negRecord.size())
      {
         String negRe = findLowNegRecord(negRecord);
         product.writeToDataBase(negRe, res);
      }
   }
}
      private String findLowNegRecord(Vector ve)
      {
         int index = 0;
         int size = 99;
         for( int i=0; i< ve.size(); i++)
         {
            StringTokenizer token = new StringTokenizer((String)ve.get(i),"/");
            if(size > token.countTokens())
            {
               index = i;
               size = token.countTokens();
            }
         }
         return (String)ve.get(index);
      }

      public void startServer()
      {
               try
               {
                        System.err.println("Initializing server: please wait." );
     // create server object
     // bind BuyerSerevr object to the rmiregistry
     String serverObjectName = ip+getMediatorKey();
     Naming.rebind( serverObjectName, this );
     System.err.println("The "+getMediatorKey()+" Server is up and running." );
               }
               catch (Throwable t)
               {
                        System.err.println(t);
                        t.printStackTrace();
```

```
                    //Ensure the application exits with an error condition.
                    System.exit(1);
                }
        }

        class RunBuyerBroker extends Thread
        {
          Buyer buy = null;
          RunBuyerBroker(Buyer buy)
          {
            super();
            this.buy = buy;
          }
          public void run()
          {
        try{
          System.out.println("creating BuyerBroker");
          BrokerProductInfo bProd = new BrokerProductInfo();
          bProd.setAprmax(product.getAprmax());
          bProd.setAprmin(product.getAprmin());
          bProd.setSupDemPastRatio(product.getSupDemPastRatio());
          bProd.setSupDemCurrRatio(product.getSupDemCurrRatio());
          bProd.setImpCurrTime(product.getImpCurrTime() );
          bProd.setImpPastTime(product.getImpPastTime());
          bProd.setImpCurrPrice(product.getImpCurrPrice());
          bProd.setCurrWar(product.getCurrWar());
          bProd.setPastWar(product.getPastWar());
          bProd.setAlpha(product.getAlpha());
          BuyerBroker bBroker =  new BuyerBroker( ip+getMediatorKey(),
        ip+getMediatorKey()+""+numberOfBroker,getSeller(),hostIp,bProd);
                    listOfBroker.add(bBroker);
        }catch(RemoteException ee){ee.printStackTrace();}

          }
        }


}
```

****

Class Seller

****

```
package seller;
/**
This class registers the seller in the environment, then waits for the message from the
buyerclone.
Once it recieves the message, then experience is retrived and seller's attitude is adjusted.
Then price ranges are calculated.  Then clones are created and send the message to the seller
clone and
waits for the results. Finally it updates the Casebase.
*/
```

```
import java.util.*;
import mediator.IMediator;
import product.*;
import broker.*;

import java.rmi.*;
import java.rmi.server.*;
import java.io.*;
import java.net.*;


public class Seller extends UnicastRemoteObject implements ISeller
{
   private IMediator myMediator = null;
   public static String ip = "//localhost/";
   public static String hostIp = "//localhost/mediatorServer";
   public String mediatorKey = null;
   public SellerGUI sellerGUI = null;
   public ProductInfo product = null;
   private String buyerBroker = null;
   private int numberOfBroker = 100;
   public Vector listOfBroker = new Vector();
   private boolean isAccess = false;
   private boolean isSeller = false;
   public boolean accept = false;
   private int numberOfClone = 0;
   private Vector negRecord = new Vector();
   private int res = 0;
   public Seller(SellerGUI gui)throws RemoteException
   {
      super();
                  sellerGUI = gui;
   }
   public String getMediatorKey()
   {
      return mediatorKey;
   }
   /**
   This method is invoked by the buyerclone to inform the seller requesting negotiation

   */
   public synchronized String setBuyerBroker(String bKey,double sDR) throws
RemoteException
   {
      while(isAccess)
      {
         try{
            wait();
         }catch(InterruptedException iEx){iEx.printStackTrace();}
      }
      isAccess = true;
      isSeller = true;
      buyerBroker = bKey;
```

```
    numberOfBroker = numberOfBroker + 1;
    product.setSupDemCurrRatio(sDR);
            product.calculateProdInfo();
        (new RunSellerBroker(this,""+mediatorKey+numberOfBroker,buyerBroker)).run();
    isAccess = false;
    notify();
    return ip+mediatorKey+numberOfBroker;
  }
  public synchronized ProductInfo getProduct()throws RemoteException
      {
    while(isAccess)
    {
      try{
        wait();
      }catch(InterruptedException iEx){iEx.printStackTrace();}
    }
    notify();
        return  product;
      }
  /**
  This method invoked by the clone to inform the seller about the deal done succesfully.
  */
  public synchronized void acceptProduct(String dispalyValue) throws RemoteException
  {
    System.out.println("in Seller.acceptProduct");
    while(isAccess)
    {
      try{
        wait();
      }catch(InterruptedException iEx){iEx.printStackTrace();}
    }
    isAccess = true;
    if( (!accept) )
     {
      accept = true;
      sellerGUI.updateResult("Product is sold",dispalyValue);
      myMediator.removeSeller(product.getProduct(),mediatorKey,true);
     }
    isAccess = false;
    notify();
    System.out.println("in Seller.acceptProduct accept = "+accept);

  }
  public synchronized boolean isAccepted() throws RemoteException
  {
    return accept;
  }
  /**
  This method invoked by the clone to inform the seller about the deal done unsuccesfully.
  */
  public synchronized void rejectProduct(String dispalyValue) throws RemoteException
  {
    while(isAccess)
    {
```

```
          try{
             wait();
             }catch(InterruptedException iEx){iEx.printStackTrace();}
       }
       isAccess = true;
       sellerGUI.updateResult("Rejected the offer by the seller",dispalyValue);
       isAccess = false;
       notify();

    }
    /**
    This method invoked by the clone to inform the seller to inform him that the product is bought
from
    other seller
    */
    public synchronized void soldProduct(String dispalyValue) throws RemoteException
    {
       while(isAccess)
       {
          try{
             wait();
                  }catch(InterruptedException iEx){iEx.printStackTrace();}
       }
       isAccess = true;
       sellerGUI.updateResult("Product is sold to other buyer",dispalyValue);
       isAccess = false;
       notify();

    }
    /**
    This method is informed by each clone about their negotiation record and also to result
    negotiation and then to update the casebase.
    */
    public void upDateDataBase(boolean negFlag, String negRec)
    {
       if(negFlag)
       {  res = 1;
          numberOfClone = -1;
          product.writeToDataBase(negRec, res);
       }
       else
       {
          negRecord.addElement(negRec);
          if(numberOfClone != -1 && numberOfClone == negRecord.size())
          {
             String negRe = findLowNegRecord(negRecord);
             product.writeToDataBase(negRe, res);
          }
       }
    }
          private String findLowNegRecord(Vector ve)
          {
             int index = 0;
             int size = 99;
```

```
            for( int i=0; i< ve.size(); i++)
            {
               StringTokenizer token = new StringTokenizer((String)ve.get(i),"/");
               if(size > token.countTokens())
               {
                  index = i;
                  size = token.countTokens();
               }
            }
            return (String)ve.get(index);
         }
         public void makeConnection(ProductInfo pro)
{
  try {
      if( getMediatorKey() == null)
      {
         // name of remote server object bound to rmi registry
         String serverObjectName = hostIp;
         // lookup hostIp remote object in rmiregistry
         product = pro;
         System.out.println("Looking up server "+hostIp);
         myMediator = (IMediator)Naming.lookup( serverObjectName );
         mediatorKey = myMediator.setSeller(product.getProduct(),
      ip,product.getName(),""+product.getCurrWar());
         System.out.println("Found server "+hostIp);
         startServer();
         sellerGUI.setTitle("Seller ID = "+getMediatorKey());
      }
   }catch ( java.rmi.ConnectException ce ) {
    System.err.println( "Connection to server failed. " +
      "Server may be temporarily unavailable." );
   }catch ( Exception e ) {
    e.printStackTrace();
    System.exit( 1 );
    }
}
         public void startServer()
         {
                  try
                  {
                           System.err.println("Initializing server: please wait." );
      // create server object
      // bind BuyerSerevr object to the rmiregistry
      String serverObjectName = ip+getMediatorKey();
      Naming.rebind( serverObjectName, this );
      System.err.println("The "+getMediatorKey()+" Server is up and running." );
                  }
                  catch (Throwable t)
                  {
                           System.err.println(t);
                           t.printStackTrace();
                           //Ensure the application exits with an error condition.
                           System.exit(1);
                  }
```

```
}
        class RunSellerBroker extends Thread
        {
          Seller sel = null;
          String sellBNum ="";
          String buyBNum = "";
          RunSellerBroker(Seller sel,String sb,String bb)
          {
            super();
            this.sel = sel;
            sellBNum = sb;
            buyBNum = bb;
          }
          public void run()
          {
       try{
          BrokerProductInfo bProd = new BrokerProductInfo();
          bProd.setAprmax(product.getAprmax());
          bProd.setAprmin(product.getAprmin());
          bProd.setSupDemPastRatio(product.getSupDemPastRatio());
          bProd.setSupDemCurrRatio(product.getSupDemCurrRatio());
          bProd.setImpCurrTime(product.getImpCurrTime() );
          bProd.setImpPastTime(product.getImpPastTime());
          bProd.setImpCurrPrice(product.getImpCurrPrice());
          bProd.setCurrWar(product.getCurrWar());
          bProd.setPastWar(product.getPastWar());
          bProd.setAlpha(product.getAlpha());
          bProd.setProduct(product.getProduct());
          SellerBroker sBroker  = new SellerBroker(ip+getMediatorKey(),
        ip+sellBNum,buyBNum,hostIp,bProd);
          listOfBroker.add(sBroker);
          isSeller = false;
       }catch(RemoteException ee){ee.printStackTrace();}

          }
        }

}
```

****

Class BuyerBroker

****

```
package broker;

/**
This class is created by the buyer to do negotiation with the seller.  This class takes
the information from the buyer and exchange messages with other clone and returns the
result of the negotiation.
*/
import java.util.*;
import seller.ISeller;
import buyer.IBuyer;
```

```
import product.*;
import mediator.*;

import java.rmi.*;
import java.rmi.server.*;
import java.io.*;
import java.net.*;

public class BuyerBroker extends UnicastRemoteObject implements IBuyerBroker
{

   private ISellerBroker mySellerBroker = null;
   private IBuyer myBuyer = null;
   private IMediator mediator = null;
   private String brokerKey = null;
   private String sellerKey = null;
   private String buyerKey = null;
   private String medKey = null;
   private BrokerProductInfo brokerProduct = null;
   private boolean sellerPriceFlag = true;
   private boolean first = true;
   private boolean proposalFlag = false;
   private String negRecord = "";
   private double alphaValue = 0.0;
   private String mes = "";
   private String displayValue = "";
   RunCalculation cal = null;

   public BuyerBroker(String oKey , String bKey,String sKey,String mKey, BrokerProductInfo
product )throws RemoteException
   {
      super();
      buyerKey = oKey;
      brokerKey = bKey;
      sellerKey = sKey;
      medKey = mKey;
      startServer();
      makeConnectionToBuyer();
      brokerProduct = product; //myBuyer.getProduct();
      makeConnectionToMediator();
      makeConnectionToSeller();

   }
   public String getBuyerKey()
   {
      return buyerKey;
   }
   public String getBrokerKey()
   {
      return brokerKey;
   }
        public String getSellerKey()
   {
      return sellerKey;
```

```
}
/**
This method is invoked by the seller to pass on the information about
its clone to inform the buyer clone
*/

public void setSeller(String sellerBrokerKey) throws RemoteException
{
   sellerKey = sellerBrokerKey;
   makeConnectionToSellerBroker();
   // mySellerBroker.setBuyer();
   mes = "Buyer Broker = "+brokerKey +" to Seller Broker = "+sellerKey +" --> ";
   displayValue = "Buyer Broker = "+brokerKey +" --> Seller Broker = "+ sellerKey;
   cal = new RunCalculation();
   cal.start();
}
/**
This method is invoked by seller clone or the buyer clone to accept an offer
*/
public boolean acceptProduct() throws RemoteException
{
     System.out.println(mes+"in ByuerBroker.acceptProduct()");
     if(!myBuyer.isAccepted())
     {
        boolean flag = mySellerBroker.acceptProduct();
        if(!flag){return false;}
        myBuyer.acceptProduct(displayValue);
        if(myBuyer.isAccepted())
        {
           caseReturn(false);
           System.out.println(mes+"accept");
           return true;
        }
     }
     return false;
}
/**
This method is invoked by the seller clone that the last offer sent by the buyer is
accepted
*/
public void accept() throws RemoteException
{
     System.out.println(mes+"in BuyerBroker.accept()");
     if(myBuyer.isAccepted())
     {
        System.out.println(mes+"in BuyerBroker.accept() product is bought");
        mySellerBroker.sold();
        caseReturn(true);
        sellerPriceFlag = false;
        return;
     }
     else
      System.out.println(mes+"waiting to accept");
}
```

```
/**
This method is invoked by either buyer clone or seller clone that the offer is rejected
*/
public void reject() throws RemoteException
{
        System.out.println(mes+"reject");
        mySellerBroker.rejectProduct();
        myBuyer.rejectProduct(displayValue);
        caseReturn(false);
}
/**
This method is invoked by seller clone informing the buyer that product is sold
*/
public void sold() throws RemoteException
{
    System.out.println(mes+"sold");
    myBuyer.soldProduct(displayValue);
}
private void caseReturn(boolean negFlag)throws RemoteException
{
    System.out.println(mes+"in BuyerBroker.caseReturn()\n negRecord = "+negRecord);
    myBuyer.upDateDataBase(negFlag, negRecord);
}
/**
This method is to continue the offers and counter offers
*/
public synchronized void setSellerPrice(double maxPrice) throws RemoteException
{
    System.out.println(mes+" in BuyerBroker.setSellerPrice althaValue = "+maxPrice);
    alphaValue = maxPrice;
    displayValue +="\nSeller proposalValue = "+maxPrice+"\n";
    if(proposalFlag)
    {
       if(maxPrice > brokerProduct.getAprmax())
       {
          reject();
          sellerPriceFlag = false;
       }
       else
          accept();
       sellerPriceFlag = false;
       first = true;
       return;
    }
    else if( maxPrice < brokerProduct.getAprmax())
    {
       sellerPriceFlag = false;
       acceptProduct();
    }
    else
       sellerPriceFlag = true;

}
```

```
        /**
        To calculate the next offer price
        */
        public double calculateBid(double alphapri, double Currbid)
        {
                        System.out.println(mes+"in BuyerBroker.calculateBid \nalphapri =
"+alphapri +"\nCurrbid = "+Currbid);
                        return alphapri + Currbid;

        }

        /**
        To calculate the Bid increment
        */
        public double calculateAlphaprime(double alpha)
        {
                double alphaprime;
          if(brokerProduct.getImpCurrPrice() == 0.0)
            brokerProduct.setImpCurrPrice(0.1);
          alphaprime = ((brokerProduct.getAprmax() - brokerProduct.getAprmin())/ 5)+ ((0.5
*brokerProduct.getImpCurrTime() *(brokerProduct.getAprmax() - brokerProduct.getAprmin()))/
(10 * brokerProduct.getImpCurrPrice()));
                return alphaprime;
        }
   public void makeConnectionToSeller()
   {
    try {
        System.err.println("Looking for Seller: please wait." );
        // name of remote server object bound to rmi registry
        String serverObjectName = getSellerKey();
        // lookup ISeller remote object
        // in rmiregistry
        ISeller mySeller = (ISeller)Naming.lookup( serverObjectName );
        System.err.println("Found Seller");
        sellerKey =
mySeller.setBuyerBroker(getBrokerKey(),brokerProduct.getSupDemCurrRatio());
      }catch ( java.rmi.ConnectException ce ) {
       System.err.println( "Connection to server failed. " +
         "Server may be temporarily unavailable." );
      }catch ( Exception e ) {
       e.printStackTrace();
       System.exit( 1 );
       }
   }
   public void makeConnectionToBuyer()
   {
    try {
        System.err.println("Looking for Buyer: please wait." );
        // name of remote server object bound to rmi registry
        String serverObjectName = getBuyerKey();
        // lookup ISeller remote object
        // in rmiregistry
        myBuyer = (IBuyer)Naming.lookup( serverObjectName );
        System.err.println("Found Buyer.");
```

```
    }catch ( java.rmi.ConnectException ce ) {
     System.err.println( "Connection to server failed. " +
       "Server may be temporarily unavailable." );
    }catch ( Exception e ) {
     e.printStackTrace();
     System.exit( 1 );
     }
}
public void makeConnectionToSellerBroker()
{
 try {
            System.err.println("Looking for SellerBroker:"+getSellerKey() +" please wait." );
            // name of remote server object bound to rmi registry
      String serverObjectName = getSellerKey();
      // lookup ISellerBroker remote object
      // in rmiregistry
      mySellerBroker = (ISellerBroker)Naming.lookup( serverObjectName );
      System.err.println("Found SellerBroker." );
    }catch ( java.rmi.ConnectException ce ) {
     System.err.println( "Connection to server failed. " +
       "Server may be temporarily unavailable." );
    }catch ( Exception e ) {
     e.printStackTrace();
     System.exit( 1 );
     }
}
      public void makeConnectionToMediator()
{
 try {
            System.err.println("Looking for medKey:"+medKey +" please wait." );
      // name of remote server object bound to rmi registry
      String serverObjectName = medKey;
      // lookup medKey remote object in rmiregistry
      System.out.println("Looking up server "+medKey);
      mediator = (IMediator)Naming.lookup( serverObjectName );
      System.out.println("Found server "+medKey);
    }catch ( java.rmi.ConnectException ce ) {
     System.err.println( "Connection to server failed. " +
       "Server may be temporarily unavailable." );
    }catch ( Exception e ) {
     e.printStackTrace();
     System.exit( 1 );
     }
}
      public void startServer()
      {
            try
            {
                  System.err.println("Initializing server: please wait." );
      // create server object
      // bind BuyerSerevr object to the rmiregistry
      String serverObjectName = getBrokerKey();
      Naming.rebind( serverObjectName, this );
      System.err.println("The Buyerbroker"+getBrokerKey()+" Server is up and running." );
```

```
                }
                catch (Throwable t)
                {
                        System.err.println(t);
                        t.printStackTrace();
                        //Ensure the application exits with an error condition.
                        System.exit(1);
                }
        }
    public void doNegotiation()throws RemoteException
    {
      if(!sellerPriceFlag)
        return;
      if(myBuyer.isAccepted())
      {
        reject();
        sellerPriceFlag = false;
        return;
      }

      System.out.println(mes+"doNegotiation");
      sellerPriceFlag = false;
      Vector count = null;
      try{
        if(first)
        {
          negRecord +=""+brokerProduct.getAprmin();
          brokerProduct.setPMin(brokerProduct.getAprmin());
          displayValue +="\nBuyer proposalValue = "+brokerProduct.getAprmin();
          mySellerBroker.setBuyerPrice(brokerProduct.getAprmin());
          System.out.println(mes+"proposal value = "+brokerProduct.getAprmin());
          first = false;
          return;
        }
        double alphaBack = calculateAlphaprime(brokerProduct.getAlpha());
        double proposal = calculateBid(alphaBack,brokerProduct.getPMin());
        negRecord +="/"+proposal;
        if( proposal > brokerProduct.getAprmax())
        {
          proposal = brokerProduct.getAprmax();
          proposalFlag = true;
        }
        brokerProduct.setPMin(proposal);
        displayValue += "\nBuyer proposalValue = "+proposal;
        mySellerBroker.setBuyerPrice(proposal);
        System.out.println(mes+"proposal value = "+proposal);

      }catch(RemoteException ee){ee.printStackTrace();}
    }
        class RunCalculation extends Thread
        {
          RunCalculation()
          {
            super();
```

```
            }
            public void run()
            {
               TimeOut tm = new TimeOut();
               tm.start();
        while(true)
        {
           try
              {
                Thread.sleep(1000);
             }catch(InterruptedException iEx){iEx.printStackTrace();}
        try{
           doNegotiation();
        }catch(RemoteException ee){ee.printStackTrace();}

        }
           }
         }
         class TimeOut extends Thread
         {
            TimeOut()
            {
               super();
            }
            public void run()
            {
        try
        {
           Thread.sleep(100000);
        }catch(InterruptedException iEx){iEx.printStackTrace();}
        try
        {
           reject();
        }catch(RemoteException ee){ee.printStackTrace();}
            }
         }

}
```

****

Class SellerBroker

****

```
package broker;

/**
This class is created by the seller to do negotiation with the buyerclone.  This class takes
the information from the seller and exchange messages with other clone and returns the
result of the negotiation.
*/

import java.util.*;
import seller.ISeller;
```

```
import buyer.IBuyer;
import product.*;
import mediator.*;

import java.rmi.*;
import java.rmi.server.*;
import java.io.*;
import java.net.*;
import javax.swing.JOptionPane;

public class SellerBroker extends UnicastRemoteObject implements ISellerBroker
{

    private IBuyerBroker myBuyerBroker = null;
    private ISeller mySeller = null;
    private IMediator mediator = null;
    private String brokerKey = null;
    private String buyerBrokerKey = null;
    private String sellerKey = null;
    private String medKey = null;
    private BrokerProductInfo brokerProduct = null;
    private boolean buyerPriceFlag = false;
    private boolean first = true;
    private boolean proposalFlag = false;
    private String negRecord = "";
    private String mes = "";
    private String displayValue = "";
    RunCalculation cal = null;

    public SellerBroker(String oKey, String sKey, String bKey,String mKey,BrokerProductInfo
prod)throws RemoteException
    {
        super();
        sellerKey = oKey;
        brokerKey = sKey;
        buyerBrokerKey = bKey;
        medKey = mKey;
        startServer();
        makeConnectionToSeller();
        brokerProduct = prod;
        makeConnectionToMediator();
        mes = "Seller Broker = "+brokerKey +" to Buyer Broker = "+buyerBrokerKey +" --> ";
        displayValue = "Seller Broker = "+brokerKey +" --> Buyer Broker = "+ buyerBrokerKey;
        cal = new RunCalculation();
        cal.start();

    }
    public String getSellerKey()
    {
        return sellerKey;
    }
    public String getBrokerKey()
    {
        return brokerKey;
```

```
}
        public String getBuyerBrokerKey()
{
    return buyerBrokerKey;
}
public void makeConnectionToSeller()
{
  try {
        System.err.println("Looking for Seller: please wait." );
        // name of remote server object bound to rmi registry
        String serverObjectName = getSellerKey();
        // lookup ISeller remote object
        // in rmiregistry
        mySeller = (ISeller)Naming.lookup( serverObjectName );
        System.err.println("Found Seller");
        makeConnectionToBuyerBroker();
    }catch ( java.rmi.ConnectException ce ) {
     System.err.println( "Connection to server failed. " +
       "Server may be temporarily unavailable." );
    }catch ( Exception e ) {
     e.printStackTrace();
     System.exit( 1 );
     }
}
public void makeConnectionToBuyerBroker()
{
  try {
                System.err.println("Looking for BuyerBroker: please wait." );
                // name of remote server object bound to rmi registry
        String serverObjectName = getBuyerBrokerKey();
        // lookup ISellerBroker remote object
        // in rmiregistry
        myBuyerBroker = (IBuyerBroker)Naming.lookup( serverObjectName );
        System.err.println("Found BuyerBroker." );
        myBuyerBroker.setSeller(getBrokerKey());
        // mediatorKey = myMediator.setBuyer(getProduct(),ip,getName());

    }catch ( java.rmi.ConnectException ce ) {
     System.err.println( "Connection to server failed. " +
       "Server may be temporarily unavailable." );
    }catch ( Exception e ) {
     e.printStackTrace();
     System.exit( 1 );
     }
}
        public void makeConnectionToMediator()
{
  try {
        // name of remote server object bound to rmi registry
        String serverObjectName = medKey;
        // lookup medKey remote object in rmiregistry
        System.out.println("Looking up server "+medKey);
        mediator = (IMediator)Naming.lookup( serverObjectName );
        System.out.println("Found server "+medKey);
```

```
    }catch ( java.rmi.ConnectException ce ) {
     System.err.println( "Connection to server failed. " +
       "Server may be temporarily unavailable." );
    }catch ( Exception e ) {
     e.printStackTrace();
     System.exit( 1 );
     }
}
        public void startServer()
        {
                try
                {
                        System.err.println("Initializing server: please wait." );
      // create server object
      // bind BuyerSerevr object to the rmiregistry
      String serverObjectName = getBrokerKey();
      Naming.rebind( serverObjectName, this );
      System.err.println("The Sellerbroker "+getBrokerKey()+" Server is up and running." );
                }
                catch (Throwable t)
                {
                        System.err.println(t);
                        t.printStackTrace();
                        //Ensure the application exits with an error condition.
                        System.exit(1);
                }
        }
/**
This method is invoked by seller clone or the buyer clone to accept an offer
*/
public boolean acceptProduct() throws RemoteException
{
     buyerPriceFlag = false;
     if(!mySeller.isAccepted())
     {
        mySeller.acceptProduct(displayValue);
        if(mySeller.isAccepted())
        {
           caseReturn(true);
           System.out.println(mes+"accept");
           return true;
        }
     }
     return false;
}
/**
This method is invoked by the buyer clone that the last offer sent by the seller is
accepted
*/
public void accept() throws RemoteException
{
     System.out.println(mes+"in SellerBroker.accept()");
     if(mySeller.isAccepted())
     {
```

```
            System.out.println(mes+"in SellerBroker.accept() product is sold ");
            myBuyerBroker.sold();
            caseReturn(false);
            return;
        }
        else
         {System.out.println(mes+"waiting to accept");}

  }
 /**
 This method is invoked by buyer clone informing the seller that product is
 bought from other seller
 */
  public void sold() throws RemoteException
  {
     System.out.println(mes+"sold");
     mySeller.soldProduct(displayValue);
  }
 /**
 This method is invoked by either buyer clone or seller clone that the offer is rejected
 */
  public void rejectProduct() throws RemoteException
  {
        System.out.println(mes+"reject");
        mySeller.rejectProduct(displayValue);
        caseReturn(false);
  }
  private void caseReturn(boolean negFlag)throws RemoteException
  {
     System.out.println(mes+"in SellerBroker.caseReturn() \n negRecord = "+negRecord);
     mySeller.upDateDataBase(negFlag, negRecord);
  }
 /**
 This method is to continue the offers and counter offers
 */
  public synchronized void setBuyerPrice(double maxPrice) throws RemoteException
  {
     System.out.println(mes+"maxPrice = "+maxPrice);
     System.out.println(mes+"Aprmin = "+brokerProduct.getAprmin());
     displayValue +="\nBuyer proposalValue = "+maxPrice+"\n";

     if(proposalFlag)
     {
        if( maxPrice < brokerProduct.getAprmin() && !mySeller.isAccepted())
        {
           myBuyerBroker.reject();
           buyerPriceFlag = false;
        }
        else
           myBuyerBroker.acceptProduct();

        buyerPriceFlag = false;
        return;
     }
```

```
        else if( maxPrice >= brokerProduct.getAprmin())
        {
            myBuyerBroker.acceptProduct();
            buyerPriceFlag = false;
        }
        else
            buyerPriceFlag = true;
    }
    public void doNegotiation()throws RemoteException
    {
        if(!buyerPriceFlag )
            return;
        if(mySeller.isAccepted())
        {
            myBuyerBroker.reject();
            buyerPriceFlag = false;
            return;
        }
        System.out.println(mes+"doNegotiation");
        buyerPriceFlag = false;
        Vector count = null;
        try{
            if(first)
            {
                negRecord +=""+brokerProduct.getAprmax();
                brokerProduct.setPMax(brokerProduct.getAprmax());
                displayValue +="\nSeller proposalValue = "+brokerProduct.getAprmax();
                myBuyerBroker.setSellerPrice(brokerProduct.getAprmax());
                System.out.println(mes+"proposal value = "+brokerProduct.getAprmax());
                first = false;
                return;
            }
            double alphaBack = calculateAlphaprime(brokerProduct.getAlpha());
            double proposal = calculateBid(alphaBack,brokerProduct.getPMax());
            negRecord +="/"+proposal;
            if( proposal < brokerProduct.getAprmin())
            {
                proposal = brokerProduct.getAprmin();
                proposalFlag = true;
            }
            brokerProduct.setPMax(proposal);
            displayValue += "\nSeller proposalValue = "+brokerProduct.getAprmax();
            myBuyerBroker.setSellerPrice(proposal);
            System.out.println(mes+"proposal value = "+proposal);

        }catch(RemoteException ee){ee.printStackTrace();}

    }

        public double calculateBid(double alphapri, double Currbid)
        {
                System.out.println(mes+"in SellerBroker.calculateBid \nalphapri = "+alphapri
+"\nCurrbid = "+Currbid);
                        return Currbid - alphapri;
```

```
}

public double calculateAlphaprime(double alpha)
{
        double alphaprime;

   if(brokerProduct.getImpCurrPrice() == 0.0)
     brokerProduct.setImpCurrPrice(0.1);
   alphaprime = ((brokerProduct.getAprmax() - brokerProduct.getAprmin())/ 5)+ ((0.5
*brokerProduct.getImpCurrTime() *(brokerProduct.getAprmax() - brokerProduct.getAprmin()))/
(10 * brokerProduct.getImpCurrPrice()));


        return alphaprime;


}


class RunCalculation extends Thread
{
   RunCalculation()
   {
      super();
   }
   public void run()
   {
      TimeOut tm = new TimeOut();
      tm.start();
while(true)
{
   try
     {
       Thread.sleep(1000);
     }catch(InterruptedException iEx){iEx.printStackTrace();}
try{
  doNegotiation();
}catch(RemoteException ee){ee.printStackTrace();}
}
   }
}

class TimeOut extends Thread
{
   TimeOut()
   {
      super();
   }
   public void run()
   {
while(true)
{
```

```
    try
       {
         Thread.sleep(100000);
      }catch(InterruptedException iEx){iEx.printStackTrace();}
 try
 {
    rejectProduct();
 }catch(RemoteException ee){ee.printStackTrace();}

 }
    }
 }


}
```

Vita Auctoris

Pratap Reddy Sathi was born in Konkuduru, India, on May 25, 1974. In 1995 Pratap received his Bachelor of Science (Microbiology and Chemistry) from Osmania University, Hyderabad, India. Pratap immigrated to Canada in 1998. In 2000 he received his Bachelor of Computer Science from University of Windsor, Ontario Canada. Pratap is in the process of completing his Masters degree in Computer Science at the University of Windsor, Ontario, Canada and plans to pursue M.B.A after two years. His current research interests include Intelligents Agents in e-commerce.