2002

An implementation of a dynamic negotiation model for competitive and cooperative agents.

Osmand N. A. Christian
University of Windsor

An Implementation of a Dynamic Negotiation Model for Competitive and Cooperative Agents

by
Osmand Christian

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
2002

Osmand Christian

Abstract

Agent technology evolved from number of disciplines such as object technology, distributed computing and artificial intelligence. Agents are supposed to autonomously engage in several types of dialogues to attain their design objectives. With the emergence of highly dynamic and uncertain e-commerce, there is currently a growing interest to develop negotiating software agents that can engage in buying and selling in a virtual marketplace. In this thesis report we propose a framework for automated dynamic negotiation among competitive and cooperative software agents. These agents are facilitated with BDI model of agency capabilities to accomplish dynamic negotiation. They make use of Case Based Reasoning techniques to learn from their previous experiences over a period of time. This framework also uses Ontology to equip agents with domain specific knowledge for reasoning purposes during dynamic negotiation. Our proposed framework suggests that it is possible to build software agents, which model limited aspects of dynamic negotiation.

To my Parents Arokiam Christian and late John Christian

Acknowledgements

I am thankful to my advisor Dr. Walid S. Saba for introducing me to intelligent agents, for his guidance and advices. I specially thank my committee members Dr. Alioune Ngom, Dr. Diana Kao and Dr. Xiaobu Yuan for their suggestions and supports. I also thank my brother Betram Christian, my friends Mr. Pratap Sathi, Mr. Nantha Kumar, Ms. Prafulla Kashireddy, Mrs. Hong Guan, Mr. Sanjay Chitte, Mr. Kannan Achan and all other friends who supported me in so many ways during my study period at the University of Windsor.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Software Agents in Electronic Commerce

Agent based computing is a recent approach to problem solving in complex heterogeneous systems. It has been attracting great deal of attention, especially among the Artificial Intelligence (AI) community. The idea of software agents however evolved from three main areas of computer science such as Artificial Intelligence, Object Technology and Distributed Systems. Since the inception of agent technology, researchers have been working on systems that could automatically perform human tasks. Software agents by being autonomous, situated in an environment, proactive and reactive with the ability to communicate among them, move from place to place, negotiate for resources and to learn makes them a suitable software for accomplishing this task in a dynamic environment [33].

Electronic Commerce, which takes place in a heterogeneous, distributed, and dynamic environment is one of many areas where software agents are used[1]. As the world looks for faster and more efficient ways of accommodating rapid and long-term solutions to everyday commercial issues, especially in business transactions, electronic commerce tends to be the best solution for number of ordinary people as well as multi-billion dollar and multi-national corporations. This awareness among customers and corporations of faster and efficient solution in commerce gave way to exponential growth of e-commerce during the past seven years.

Most of the electronic purchases today are non-automated. They can only support non-interactive buying and selling retail market or auction based systems. Although the information on products and vendors are easily accessible and orders and payments are dealt with electronically, humans are still involved in the loop at every stage of the

[1]See Appendix A for details on Electronic Commerce

buying process. Buyers are still responsible for collecting and interpreting information on products, evaluating merchants, involving in negotiation process with merchants and ultimately finishing the deal by providing purchase and payment information [28]. Furthermore, customers of today's e-commerce are faced with problems of too much information on products and frequent change of site content.

There are few applications available over the Internet to assist customers on 'what to buy' and 'whom to buy the product from' [28]. However, there are no applications available yet that could perform 'automated negotiation' for customers in e-commerce. Buying and selling agents can be used to eliminate most of the problems in today's e-commerce. These software agents can collect and filter product information, evaluate merchants based on user's preferences, negotiate in an uncertain buying-selling environment and finally finish the deal for the user on his/her behalf and save time for the user [34]. In research, there are few models attempted to simulate automated negotiation in buying and selling environment. However, none of these models had taken into consideration of the environment changes that could affect the on going negotiation process in the system. An environment change that could affect the negotiation process for instance is supply and demand ratio of a product for which agents are negotiating in the marketplace.

1.2 Principle of Automated Dynamic Negotiation

When faced with the need to reach agreements on a variety of issues, humans make use of negotiation process. Similarly, automated negotiation can become a fundamental operation for shopping agents in e-commerce. Negotiation is defined as the 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. Real world negotiations in general accrue transaction costs and time that may be too much for both merchants and consumers alike. A good automated negotiation can both save time and find better deals in the complex and uncertain business environment [39].

Research area that merges negotiation with software-agents is the broad field of Multi Agent System (MAS)[2]. MAS and Distributed Problem Solving (DPS) are part of Distributed Artificial Intelligence (DAI). Early DAI work modeled negotiations as DPS and assumed a high degree of joint *cooperation* among agents in order to achieve a common goal. In MAS, there is no global control, no globally consistent knowledge, and no globally shared goals. They are concerned with coordinating intelligent behaviour within a collection of autonomous (possibly heterogeneous) intelligent agents. MAS assume total self-interest and a high degree of *competition* among agents during negotiations for limited resources [28]. The agents in a cooperative model, in general, share at least one aspect of their goal. In contrast, agents in a competitive model hide their goal from the opponents [37]. This competitive behaviour of MAS seems to best suites our needs during negotiation in e-commerce environment.

There are two important theorems exists on negotiation, viz., Game theory and Epistemic Logic[3]. Although various disciplines have proposed different theorems on negotiation, it is clear that negotiation theory[4] covers a wide range of phenomena encompassing different approaches such as Artificial Intelligence, Social Psychology, and Game theory [29]. Negotiation research can be considered to deal with three broad topics [4],

1. *Negotiation Protocol*: set of rules, which govern the negotiation processes.
2. *Negotiation Objects*: range of issues over which agreements are to be reached.
3. *Decision Making Model*: Decision-making apparatus used to achieve negotiation objectives.

However, the relative importance of these three topics may vary according to the negotiation and environmental context.

[2] See Appendix B for details on MAS
[3] See Appendix C for details on Game theory and Epistemic Logic
[4] See Appendix C for details on Negotiation Theory

The minimum capabilities required for an automated negotiation process is to propose some part of the agreement space as being acceptable and to respond to such a proposal indicating whether it is acceptable. That is a 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. When electronic buying-selling agents are equipped with negotiation tactics, reasoning capabilities and adaptability to uncertain environment changes, they can negotiate without any human interaction at any virtual marketplace. The autonomous, proactive and reactive nature of software agents makes them suitable candidates for automated negotiation in dynamic and complex e-commerce transaction. To be competitive in today's e-commerce marketplace, buying and selling agents need to act rationally and intelligently, taking into account of the environmental changes that occurs in e-commerce environment.

1.3 The Problem Domain

1.3.1 Problems in e-commerce negotiation

Formalization of a negotiation process can be very complex as there are many protocols and properties to be considered. The general properties desirable for negotiation mechanism are computational efficiency, communication efficiency, distribution of computation and individual rationality. The former three issues above pose major software engineering challenges. Individual rationality however is more complex and challenging as it depends on the following parameters [24]

1. Cardinality of negotiation (one to one, one to many and many to many)
2. Agent characteristics (role, knowledge, commitment)
3. Environment and goods characteristics (static or dynamic[5], private-public value)
4. Event parameters (importance of time, schedules)
5. Information parameters (price quotes, transaction history (experience))

[5] dynamicity of the environment itself

These parameters may vary from domain to domain and as a result, in most cases, negotiation strategies and tactics are completely domain dependant. Besides the challenges faced in negotiation mechanism, challenges in automating negotiation in e-commerce includes [15]:

1. It is 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 complex process and there is no universally accepted negotiation technique. Some of the important negotiation models for e-commerce in research include [5, 15, 30, 37, 38, 41, 42]. Especially, [37 and 38] presents a unique solution to automated negotiation in e-commerce. Agents in this model takes into account of prior experiences of purchasing similar products, domain specific information of products and attitude of buyers and sellers during negotiation process. Although the above models present interesting approaches towards solving the automated negotiation problem that exist in e-commerce systems, none of them have considered the effect of dynamicity of the environment on those negotiation processes they have presented.

1.3.2 Thesis Contribution

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

"It is possible to build dynamic negotiation for competitive and cooperative autonomous agent systems with the limited aspects of negotiation using current software technologies."

This thesis proposes a framework for automated dynamic negotiation among competitive and cooperative software agents in e-commerce environment. The agents in this framework are equipped with Belief, Desire and Intention decision-making model for dynamic negotiation[6]. Moreover, the agents in this framework tend to learn from their experience over a period of time. Learning from experience in these agents is modeled using case based reasoning (CBR)[7]. This framework also uses Ontology to equip buyer and seller software agents with domain specific knowledge for reasoning purposes during negotiation.

1.4 Outline of the Proposed Model

The framework proposed in this thesis is a prototype of a virtual marketplace where buying and selling agents autonomously engage in negotiation on behalf of their clients. Although this negotiation framework shares several common features with number of existing approaches to negotiation as in [21, 36, 37, 41], this framework uniquely explores dynamic negotiation capabilities of agents when the environment change affects the negotiation process itself.

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

> *"Agents make dynamic decisions during the negotiation based on BDI model of agency to compensate the changes that occur in the uncertain e-commerce environment".*

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

---

[6] See Appendix E for details on BDI model of agency
[7] See Appendix D for details on CBR

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 and dynamic negotiation of this framework is presented. Chapter 4 discusses the implementation details of the proposed model. In chapter 5, evaluation of the proposed model is discussed. Chapter 6 discusses the conclusion and future work.

Chapter 2

An Overview of Existing Negotiation Models

This chapter reviews some of the important related models and approaches that enable automated negotiation in e-commerce. Automated negotiation has become a promising area with the exponential growth of e-commerce during the past few years. Parameters involved in negotiation and the high degree of uncertainty that exist in a virtual marketplace pose many challenges when creating an e-commerce system. Researchers have presented several e-commerce models, taking into account of number of negotiation parameters and the dynamics of the e-commerce environment.

Current approaches suggested by researchers in automated negotiation for e-commerce include KASBAH [5], Negotiating with experience [37], MAGALE [41], Negotiation model for multiple transaction factors and learning in e-commerce [15], Experienced agents with attitude in virtual marketplace [36], Negotiating with an attitude in a virtual marketplace [38], Computational model for online agent negotiation [42] and Agent negotiation as fuzzy constraint processing [17]. Surprisingly, none of these models pay close attention to e-commerce environment changes, even when these changes may affect the negotiation process of the participating agents. This chapter briefly reviews and evaluates the above approaches and finally presents a dynamic decision making theorem called BDI model of agency [21]. This BDI model of agency theorem is of high importance in solving dynamic environment negotiation problems in an uncertain e-commerce environment.

2.1 KASBAH

Kasbah is an electronic agent marketplace, born in MIT Media Laboratory. It is a Web-based multi-agent classified ad system where users create buying and selling agents to help transact products. Buying and selling agents in Kasbah negotiate to buy and sell goods and services on behalf of their users. This e-commerce system has three major

components. They are Front-end, Back-end and Auxiliary components. Front-end is a web interface that handles user interface, Back-end is the actual marketplace engine where agents operate and interact with each other and the last component is the generator of display files and notification for the user [5].

Negotiation strategies in this model are predetermined and a user is allowed to select a negotiation strategy when he/she creates an agent. A buyer or a seller also defines the goal of the respective agent by specifying desired price, lowest acceptable price and desired date to buy or sell. There are three predetermined strategies in Kasbah. They are anxious, cool-headed and greedy which are linear, quadratic and cubic decay functions respectively as shown in figure 2.1 [30]:

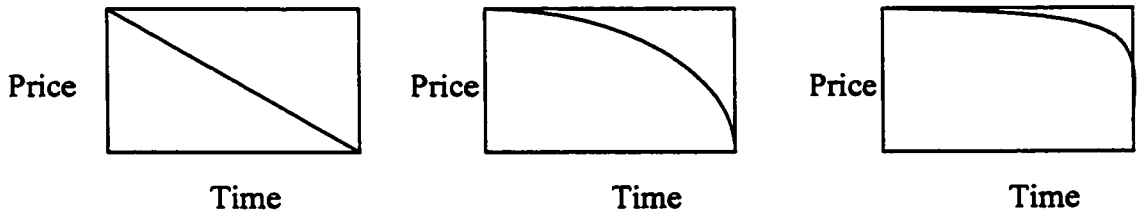

Figure 2.1: Price change in time for selling agents in Kasbah
(Left to right: anxious, cool-headed and greedy)

During the process of negotiation, if the opponent accepts the offer, the negotiation is terminated. Otherwise, in case of seller agent, the agent would lower the price till it reaches the lowest acceptable price by following a chosen negotiation strategy. Intuitively, a buyer agent would work the opposite way till it reaches the maximum acceptable price with a chosen negotiation strategy.

Kasbah was one of the first systems that tried to imitate real world negotiation using time, money and various negotiation strategies. Although it was a novel start that was to revolutionize business in the near future, it had its own drawbacks. Agents in Kasbah are simple and there is no learning mechanism in the system. Furthermore, their decision strategies are limited and decision selection is not autonomous. Moreover, only *price* drives negotiation in Kasbah [15, 26]. Negotiation can be done over multiple parameters,

where agents can make concessions over one or more issues [4]. In summary, Kasbah doesn't support negotiation on multiple attributes, it doesn't have autonomous negotiation strategy and there is no learning mechanism in this system.

2.2 Negotiating with experience

This model uses Case-Based Reasoning techniques to develop negotiation strategies for current situation. This experience based negotiation framework provides adaptive negotiation strategies that can be generated dynamically and are context-sensitive. Architecture of this system consists of three main parts. The first part is a *Case-Based Negotiator*, which assists the user while negotiating with opponent agents. It matches current negotiation scenario with previous successful negotiation cases and provides appropriate counter-offers for the user based on the best-matched negotiation case. Second part is a *Case Browser* that allows user to browse a previous negotiation case repository using various queries and the third part is a *Case Maintenance component* that allows negotiation experts to moderate, maintain and to update case repository [37]. The agents in this model are rational and the negotiation process here is strictly monotonic.

A web-based used car-trading negotiation model is presented as an example in this model. Initially, the case base system of this model is populated with number of cases that are relevant to the product domain of the application. A user creates an agent in the system by selecting the car, buy/sell information, budget, sex, age etc. Once the agent is created, negotiation starts with the opponent agent. A concession match filter is used in finding the best-matched case. This concession matcher tries to find matches between the offer and counteroffer of the previous negotiation cases in case base and the offer and counteroffer of the current negotiation. During a decision-making moment in the negotiation process, the *case base negotiator* retrieves relevant cases from the case based system with the help of concession filter and adapts strategy from the best-matched negotiation case to generate an offer or counter-offer. If there is no match found in the case base that is similar to present case, a predefined strategy is used [37].

This model presents a very interesting approach that tries to imitate the aspects of human buying process. However, the work is limited to only one particular domain (used car-trading) and negotiation in this model is based on single attribute, namely *price*. Moreover, the attitudes of the buyers and sellers are not taken into consideration. Attitude of a user may include many factors. Some of the obvious factors are importance of time (urgency), importance of price (price consciousness) and commitment of the user for the given transaction. Attitude of the user plays a major role in transactions along with experiences. The work is mostly concentrated on matching of cases using the concession filter technique with less focus on and no consideration of actual proposal generation. Also, learning from failure is not considered in this model.

2.3 MAGALE - Multi-Agent Architecture for Adaptive Learning Environment

Marketplace for learning resources such as advice and tutoring are the main focus of this model. A user who possesses knowledge or resources becomes seller and a user who seeks help or advice becomes buyer in this marketplace. The agents in this system decide how to increase or decrease price for resources based on their user's preferences. Some of the user's preferences considered in this model are urgency of the user's current work, impotence of money to the user and the user's risk behavior [41]. There is asynchronous and synchronous information exchange in this system. Asynchronous information includes web pages and FAQ entries where a buyer would pay to access a site to obtain needed recourses. Synchronous information may include an online help session via chat, telephone or collaboration environment, where real-time live contact is made between the buyer and a seller. Synchronous information exchange needs negotiation since many factors may play a role in dynamically determining the price of the service. How urgent the buyer needs help and how busy the helpers in the system are some of the factors that dynamically determine the price of the service.

The agents in MAGALE represent users. They maintain information about user's goals, preferences and knowledge. When user needs help, the user's agent contacts a centralized matchmaker who knows which users are online. These agents negotiate with each other about the price and when a deal is made they inform their users. Agents make decisions on behalf of their users to find a better deal. They involve in offer and counteroffer iteratively based on users' preferences. As the negotiation is iterative, it allows the negotiating agents to change their preferences dynamically between each offer and counteroffer. This creates a high degree of uncertainty in the marketplace. An influence diagram is used to model uncertain variables and the decision making process of this system as in figure 2.2.

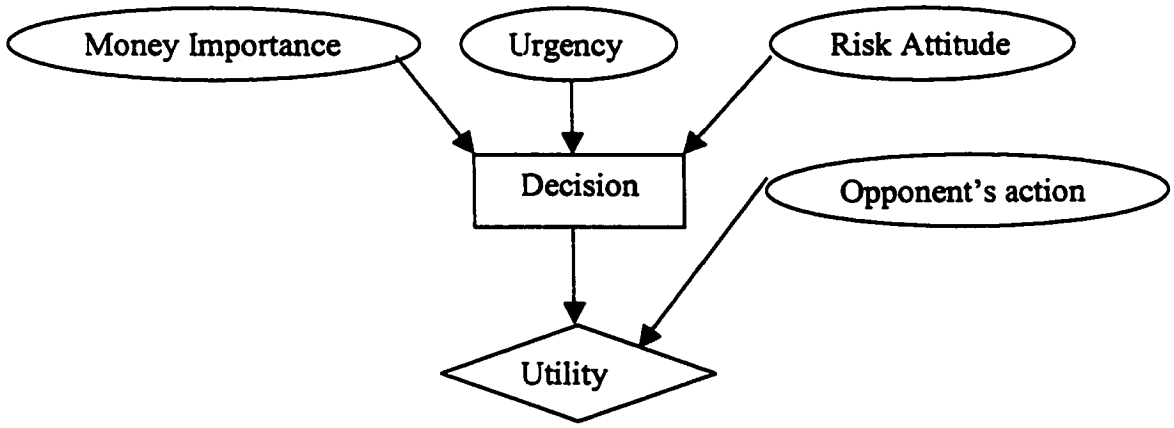

Figure 2.2: Influence Diagram for the decision model in MAGALE [41]

An influence diagram with *decision* node representing the choices available to the user, *chance* (circle) node carrying probabilistic information of the uncertainty in the environment and opponent, and *value* (diamond) node representing utility, help find the optimal solution for the problem. Moreover, influence diagram in this model helps agents to predict the opponent's reaction during negotiation process.

User's preferences such as importance of money, urgency and risk behavior plays a major role in the decision-making process in MAGALE. For instance, a 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 from counter proposing in fear of losing the deal. The state of a negotiating agent in this system could be in any of the following states: Accept, Reject or Counter propose [41].

Probabilistic influence diagram that helps model opponent agents to predict their behavior during the negotiation process and the idea of user's preferences used in decision making are genuine in this model. However, the accuracy of the probabilistic influence diagram, in terms of the utility users may receive, is still questionable. Besides, the complexity of this process may be very expensive when there is large number of participants involved. Moreover, this system doesn't consider simultaneous negotiations when multiple buyers and sellers are present in the system for the same resources.

2.4 Negotiation Model for Multiple Transaction Factors and Learning in E-commerce

This model presents a flexible negotiation system for agent-based e-commerce systems. Agents in this model negotiate with number of opponents simultaneously over several attributes. Negotiation process takes place between CAs (Customer Agents) and SAs (Seller Agents) in this model. They both negotiate to obtain a better deal on behalf of the user's goal. To enable CA to negotiate with more than one SA and vise versa, replicas of CAs and SAs are created in this model. Each replica of a CA negotiates with a unique SA replica and each replica of a SA negotiates with a unique CA replica. CA replicas and SA replicas negotiate by passing offers and counteroffers to reach a deal in the system. The replicas in this model learn by means of Black Board approach during negotiation. All replicas of an agent notify their progress in negotiation to their parent agent (CA/SA). Parent agents keep all their replica's progress information in a Black Board. All replicas of an agent have access to the progress information of other replicas of the same parent agent, through their parent agent's Black Board. This Black Board approach helps replicas to learn and negotiate better in two ways. First, it enables replicas to compare and analyze negotiation strategies of all the Sellers in the marketplace. Second, it enables

replicas to understand and analyze a negotiation trend of all the SAs in real time. Using this knowledge, agents can come up with new negotiation strategies from time to time and also it helps agents to select its own strategy safely and conveniently [15].

An ontology describes objects in a way that is semantically meaningful and non ambiguous to software agents [15]. This system uses ontology to represent knowledge and it is made open to both buyers and sellers to handle negotiation process. It means both CAs and SAs in this system can easily add item-specific attributes and their personalized values for those attributes. This makes it possible for agents to consider not only price but also other attributes during their negotiation in this model. CAs and SAs input rules for all the attributes they consider during negotiation. These rules are stored in a *history* component of this model. Note that learning in this system takes place by means of rule-based learning strategy. The rule-based knowledge from *history* is referred by CAs and SAs during negotiation process so that the system becomes automated and adaptable to any attributes considered during negotiation.

Creation of replicas (agent cloning) makes execution of tasks faster in this model. However, in real time e-commerce negotiations where the negotiation attributes are very dynamic and where agents tend to hide their information, the agents in this model would fail to learn form opponent's strategies.

2.5 Agent Negotiation in a virtual marketplace

Automated negotiation in dynamic environment is the main concern in this model. This model consists of buyers and sellers, Ontology and a Case Base Reasoning system. It presents a virtual market place with experienced based buying and selling negotiating agents. Importance is stressed on the mental attitude of the user and in turn it reflects on the automatically negotiating buying and selling agents in finding the price range for the product. Mental attitude is comprised of importance of price, importance of time and commitment of the user. Just as in real world negotiation how a buyer or a seller would

hide his/her mental attitude to get the best deal, these agents in this marketplace also hide their information from opponent agents to find the best deal [36].

Each existing product in the virtual marketplace has a public price range in the ontology. This is to equip the buying and selling agents with the domain specific information they need for negotiation. Moreover, the Case Based Reasoning system holds the unique past experiences in the form of negotiation records. A best matched past negotiation record from the case based reasoning system is used to assist the negotiation in this model by adjusting the actual attitude provided by a buyer or seller. An agent enters the market place with a maximum and minimum price range, finds the seller and negotiates on behalf of the user. Notably, agent's attitude plays an important role in determining how a negotiation will proceed in this model. Given an agent, it can be either in $Done^{+}$ or $Done^{-}$ or in $Done^{0}$ state representing, negotiation terminated successfully, agent failed to reach an agreement or negotiation is in progress, respectively. Regardless of the outcome, the results are saved as experiences in a case base for future use [36].

Mental attitude of the agent and the use of past negotiation experience makes this model a unique e-commerce system. However, parameters that can be considered for mental attitude may be numerous. Accordance with the parameters considered, the functions used to arrive at the minimum-maximum price range should be given a thorough study to bring this model into a real world e-commerce environment. Moreover, the change in the e-commerce environment such as supply demand ratio change or public price range change is not handled in this model.

2.6 Negotiating with an attitude in a virtual marketplace

The focus of this model is to create competitive and corporative negotiating agents in a single mental state model. In a competitive dialogue type such as in negotiation between buying and selling agents, participants have a fixed goal and a well-defined utility function that is used to measure their progress towards achieving their goal. In a cooperative dialogue type, where agents corporately decide on a course or courses of

action, participating agents have no fixed initial commitment to any potential course of action. This model proves that regardless of the dialogue type, agent dialogues are a function of participants' mental state, i.e., the participants' goals and attitude.

This virtual marketplace model assumes agent's mental attitude as a tuple that contains three attributes. They are importance of time, importance of price and commitment of a user, which takes numerical values between intervals of zero and one. A model becomes cooperative when all participants agree on at least one of these attributes. But a competitive model does not assume any such constrain [37]. In fact, it hides its mental attitude from the opponent agent to gain maximum utility. However, a competitive model's participants may agree on their mental attitude. For instance, if **b** is a buyer agent with the mental attitude of <x1, x2, 1.0> and **s** is the seller agent with the mental attitude of <x3, x4, 1.0>, they both are highly committed towards buying and selling the product as their commitment level is 1.0.

In the above scenario, the negotiation process becomes more cooperative than competitive because both the buyer and seller are highly committed to buying and selling the product. This claim is mathematically proved in this model using the buyer and seller agent's maximum and minimum price range functions as follows:

$$\text{APR}^{\mathbf{b}}(\text{PPR}(\text{product}),\ (x1, x2, 1.0),\ \text{SDR})$$
$$= [< pmin + (t)(pmax\text{-}pmin)/10 >, < pmax - (pmax)(p)(1\text{-}c)/10 + pmax(1\text{-}SDR)e >]$$
$$= [< pmin + (t)(pmax\text{-}pmin)/10 >, < pmax + pmax(1\text{-}SDR)e >]$$

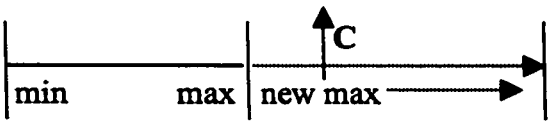

Figure 2.3: Effect of commitment level for buyer

$$\text{APR}^{\mathbf{s}}(\text{PPR}(\text{product}),\ (x3, x4, 1.0),\ \text{SDR})$$

$= [< pmin + (pmin)(p)(1\text{-}c)/10 + pmin(1\text{-}SDR)e >, <pmax\text{–}(t)(pmax\text{-}pmin) / 10 >]$

$= [< pmin + pmin(1\text{-}SDR)e >, <pmax\text{–}(t)(pmax\text{-}pmin) / 10 >]$

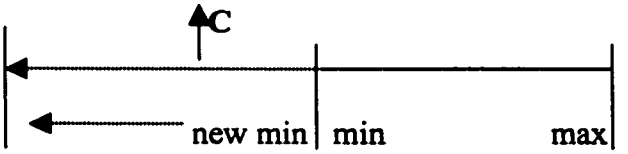

Figure 2.3: Effect of commitment level for seller

where *APR* is the Agent's Price Range,

PPR is the public price range for a product,

pmin is the public minimum price for the product,

pmax is the public maximum price for the product,

C is the agent's commitment to buy the product,

t is the agent's importance of time,

p is the agent's importance of price, and

SDR is the supply demand ratio in the market for the product.

As we can also see in the figures above, the more the agents agree on the commitment attributes, the more buyer agent would increase the maximum price and the seller agent decrease the minimum price. This can be interpreted as agents cooperating as this leads to more successful deals [38]. Therefore, agents cooperate when hidden mental attitudes of the agents agree in this model.

2.7 Computational model for online agent negotiation

The negotiation process of this model is based on the internal beliefs of the participating agents. Unlike other proposed models, negotiation process is considered to be a sequential decision-making process in this model. In every negotiation iteration (offer and counter-offer), an agent checks the history of the negotiation process, updates its beliefs about its opponents and then tries to maximize its expected payoff based on its own subjective beliefs. Opponent agent's action set is the basis for an agent's belief in

this model. However, agents can be uncertain about what action their opponents might take. Therefore uncertainty in this model is in the process of negotiation itself. This model uses BDI model of agency to solved uncertainty in opponent agents action set. Admittedly, the only observable action of an opponent in this system is the price offer [42].

The buyer and seller have their own reservation prices for a product in this model. In each negotiation iteration, if the buyer's offer is no less than the seller's then the negotiation ends. Otherwise the negotiation goes on to next iteration until the maximum time of the agent to negotiate is reached. Uncertainty in the opponent agent's next action is taken into consideration when an agent updates its belief at the end of every iteration. The next offer or a counter offer of an agent is based on which uncertainty of the opponent agent was chosen after the last negotiation iteration. These uncertainties are based on the time left for the agent to finish negotiation. In case of a buyer agent, its belief may be one of the following [42]:

- the less the time left, the more buyer may believe that the seller will not change the current offer.
- the buyer may not believe that the seller is offering a reasonable price.
- the less the time left, the more buyer may believe that the seller will decrease the current offer.

One of the difficulties presented to a negotiation agent on the Internet is that it has little information about its opponents. This is the main reason behind incorporating belief mechanism in this model. This model shows that depending on different internal beliefs an agent may behave differently, just like human beings. However, there is an obvious problem attached to this model. Based on the belief selection, an agent may choose to be hard on the opponent which could result in making no profit or it may be too easy on the opponent and make less profit when making a deal. If this model could somehow incorporate some trade-off mechanism to handle this problem, it could produce very interesting results.

2.8 Agent Negotiation as Fuzzy Constraint Processing

A general framework for agent negotiation based on *fuzzy constraint processing* is presented in [17]. As an example, a negotiation process between Police institution (CA) and Badge makers (SA) are also presented in this paper. Negotiation in this example considers issues like price, time of delivery and the quality of Badge. The attributes involved in this negotiation may be clear or fuzzy. When a customer agent doesn't have an exact idea of the price he/she prefers or the quality he/she requires, the customer will give an implicit requirement of "short delivery time", "cheap price", and "high quality". A user can input these fuzzy predicates, which are "short", "cheap" and "high" in numerical form to start the negotiation.

Agent negotiation is formulated as a Distributed Fuzzy Constrain Satisfaction Problem (DFCSP) in this framework. Solution to a DFCSP is achieved by fining the consistent solution that satisfies all constraints in DFCSP network. Solution to agent negotiation problem is closely related to DFCSP, as agent negotiation also has to come up with a mutually acceptable agreement among two or more agents with the fuzzy attributes in mind. When modeling negotiation problem in DFCSP, the goal of the negotiation and the utility function are represented as constraints. Moreover, the problem of agent negotiation in this model is looked at as finding a consistent solution that satisfies all constraints with maximum satisfaction between buyer and seller agents. Any negotiating agents usually aim for the maximum profit. However, agents in DFCSP also need to find the consistent solution with maximum satisfaction in all the fuzzy constraint attributes. This could produce conflicting interests in finding the maximum profitable solution. Therefore, negotiation strategy in this model is viewed as fuzzy constraint processing that resolves conflict and generates optimal solution [17]. When there is a conflict during negotiation, agents use concession relaxation or reconfiguration to find a solution.

This model incorporates three types of concession strategy to determine the offer and counter offer to move the negotiation forward. They are:

- *Fixed Concession Strategy*: where the next offer is assigned by considering the urgency of the negotiating agent
- *Reaction Concession Strategy*: where negotiation agent computes the next offer based on the last offer and the one before of the opponent agent.
- *Flexible Concession Strategy*: where the negotiation agent computes the next offer based on combination of *Fixed Concession strategy* and *Reaction Concession Strategy.*

This is a very interesting approach that tries to model fuzzy requirements and fuzzy preferences involved in negotiation into a Distributed Fuzzy Constraint Satisfaction model. However, there is no learning provided to the buying and selling agents in this system and the fuzzy constraints considered in the model are limited to just three attributes.

2.9 Belief-Desire-Intention model of agency

Most of the conventional software applications are designed for static world. These conventional software are assumed to work with perfect knowledge, meaning that they have all the information they need to make their decisions. However, in the real world, these systems are embedded in dynamic environments. Therefore, when it comes to real world issues, they have only *partial information* available for them to make any decision i.e., their access to dynamic information is limited. Moreover, the systems in existence don't have unlimited computational recourses [21].

Belief, Desire and Intension (BDI) decision-making model is interested in solving dynamic and uncertain environment problems. BDI model has become the best-known and best-studied model of practical reasoning agents. There are several successful applications exist based on BDI model. Fault diagnosis system for space shuttle and factory process control system are two examples where BDI model of agency is used [2].

Belief represents the knowledge of the world. Computationally, Believes are some way of representing the state of the world. For example, belief in a BDI system could be a value of a variable or tupples of a rational database. Hence, Belief represents information about the world. Belief is needed because the world changes and we need to remember the past events. The reason why we need to remember the past events is because if I want to get somewhere for example, I need to know where I am right now in order to find out how I can get there.

Desire, or more commonly, the Goal. Computationally, it may be a value of a variable or a symbolic expression in some logic. The important point is that a Goal should represent some desired end state. Conventional systems also have desired end state but they are 'task oriented' than goal oriented. This means the system cannot automatically recover from failures. For example, the reason we recover from a missed train is because we know where we are (through our believes) and we remember to where we want to go (through our Goals). Task oriented conventional software would fail in this above situation but BDI model would not.

Belief and desire enables to decide on a plan to achieve the goal. However, in a dynamic environment, where there can be a change in the environment that could affect the achievement of the goal, what should we do? Classical decision theory states that we should always replan when there is a change in the environment. In contrast, the conventional system goes on executing the tasks with no consideration to the changes in the environment. There are problems associated with both the ideas. A system cannot ignore the changes and execute the tasks in a dynamic environment nor it can replan for every single change in the environment because of the limited resources.

The third component of the BDI model, **Intention** states that 'the system needs to commit to the plans and sub goals it adopts but it must also be capable of reconsidering these adopted plans at appropriate (crucial) moments. Computationally, Intensions may be simply be a set of executing threads in a process that can be appropriately interrupted

upon receiving feedback from the possibly changing world. For example a *Flight Scheduler agent* which schedules arrival time of the flights on the runway in an airport may have number of threads running for each flights. As the weather changes (environment changes) the process should be appropriately interrupted so that the process can replan and inform the respective flight to make sure that flight can still reach the runway at the expected time [2].

A negotiation in a real world, either cooperatively or competitively, takes place in an uncertain and dynamic environment. This BDI model of agency theory gives us a realistic decision solution to such dynamic environment negotiation problems.

Chapter 3

Dynamic Negotiation for Competitive and Cooperative Agents

A framework for automated dynamic negotiation in e-commerce is presented in this chapter. The software agents involved in this framework are both competitive and cooperative. An agent's mental attitude towards buying or selling a product, general price range, buying-selling experience with similar product and supply-demand ratio are few important parameters that determine how a negotiation should proceed in an e-commerce environment.

In today's e-commerce environment, buyer and seller agents can dynamically enter and leave the marketplace with external knowledge[8] of a certain product. The agents that are already negotiating for the same product in the system should somehow become aware of any external change occurred for that product. Becoming aware of the changes that happened inside[9] or outside the system would affect the negotiation process of the agents who are already negotiating for that product. Capturing and compensating for the changes by changing the strategy of the agents who are already negotiating in the system is what is considered as dynamic negotiation in this model.

A negotiation model, either it is static or it is dynamic, needs a negotiation protocol. A negotiation model should know the issues over which the agreements are to be reached and should also have a decision making model to reason during the negotiation process [4]. The negotiation protocol and the negotiation objects considered in this framework are same as in [37]. The decision making model in this framework however is more sophisticated and handles dynamic events that happens inside and outside the e-commerce environment by using the BDI model of agency principle.

---

[8] An external knowledge of a certain product can be of sudden price drop for a product due to war on terrorism.

[9] An inside or internal change for instance is the supply demand ratio change due to agents entering and leaving the marketplace.

As explained in Chapter 1, this research is concerned with dynamic negotiation model for competitive and cooperative agents in an e-commerce system.

3.1 Negotiation Protocol

Agents negotiating for goods need to interact or communicate with each other in an e-commerce marketplace. A negotiation protocol is required to handle the communication safely and efficiently among these agents. Interactions between agents are modeled as offers and counteroffers in this framework. Offers and counteroffers terminate when a deal is made successfully or unsuccessfully between buyer and seller agents. A deal is made successfully when buyer agent's maximum price is grater than or equal to seller agent's offered minimum price. An unsuccessful deal occurs when buyer's maximum falls short of seller's minimum.

3.2 Negotiation Issues

Unlike many other proposed e-commerce models, [37] explores negotiation over multiple issues. Negotiation issues considered in this framework include price, warranty, supply-demand ratio and attitude of the user. Attitude of a user is composed of importance of time, importance of price and commitment attributes representing urgency of the user, importance of money and user's level of commitment towards buying or selling the product, respectively. Along with the importance of price and warranty, other factors such as urgency and commitment also play a major role in negotiating for a product. Among many other factors that could affect the negotiation, dynamic and uncertain factors like change in supply-demand ratio and general raise or fall in price range of a product is also considered in this dynamic negotiation model.

3.3 Decision Making Model

Previous similar buying-selling experiences, market conditions, general price range of a product, user's mental attitude and what maximum or minimum a user is willing go are taken into consideration when deciding an agent's maximum-minimum price range. Moreover, the decision-making model we employ in this framework takes into account of

the dynamic changes that affects the negotiation process. We follow the idea of BDI model of agency to accomplish this dynamic decision making task in our system. *Intention* mechanism of the BDI model is used to re plan (re compute the maximum and minimum price of an agent) when there is a change inside or outside the system so that the agents will be competitive in the marketplace.

3.4 Overview of the Framework

The interaction between agents in this e-commerce model is many to many; one buyer may simultaneously negotiate with as many sellers and one seller may simultaneously negotiate with as many buyers in the marketplace for a product. Buying and selling agents dynamically negotiate on behalf of their users in this model. This section briefly discusses the basic components and negotiation process of this dynamic model.

3.4.1 Basic components of the Framework

This framework consists of list of buyers and sellers (list of Buyeers&Sellers), Ontology and a Case Base Reasoning system. List of Buyers&Sellers helps buyer or seller agents to find their opponents. It also enables the system to dynamically find the supply demand ratio for any product in the market.

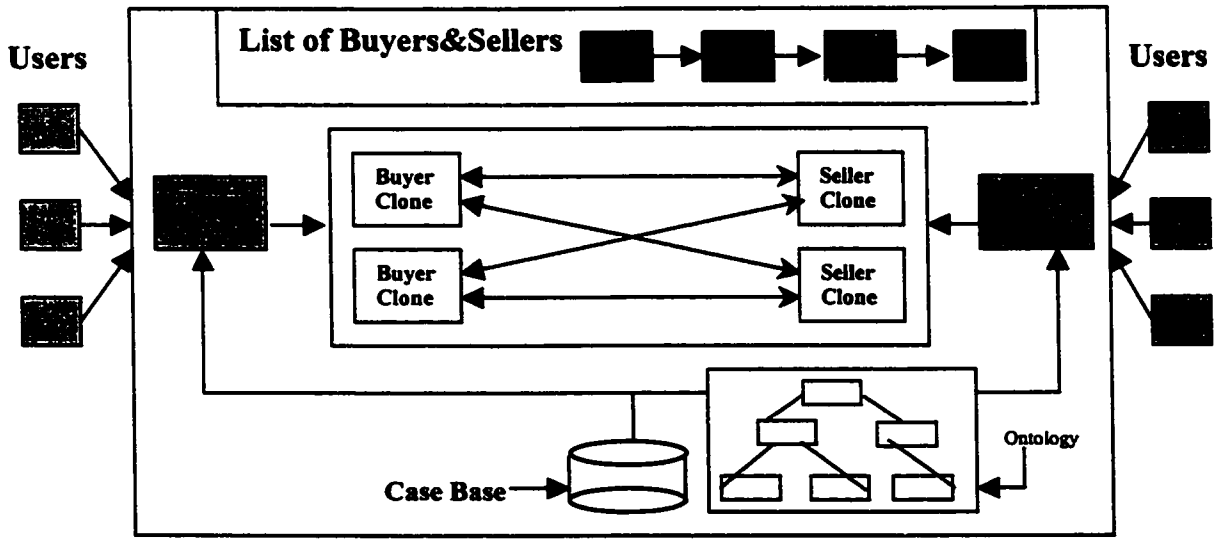

Figure 3.1: Basic components in of the framework

Each buyer and seller agent in this system has a unique name to identify each other during the negotiation process. Ontology provides the domain specific knowledge of a product in the market. Both buyer and seller have access to this domain specific information of products that are available in the marketplace. Case based reasoning technique is used in this model to learn from past negotiation experiences and to use them in current negotiation situation. Buyers and sellers are given access to their past negotiation experiences to learn and to reuse those experiences. Figure 3.1 shows the high level view of this framework.

3.4.2 Overview of the dynamic negotiation process

Negotiation process starts when a user creates a buyer or a seller agent in the marketplace. This agent enters the marketplace with user's mental attitude, his/her maximum and minimum price range and the expected or givable warranty. Once the agent enters the marketplace, it gets registered with the list of Buyers&Sellers. All the agents in the marketplace are maintained dynamically[10] in the list of Buyers&Sellers. It is buyers who initiate the negotiation with sellers in this framework. To maintain low level of complexity, the scenario where a seller initiates the negotiation is avoided in this framework.

An overview of this process, where a buyer is entering the marketplace is described bellow:

- A buyer creates buyer agent **b** with attitude, warranty and max-min price
- **b** enters the marketplace and registers in list of Buyers&Sellers
- **b** retrieves the publicly available price range from the Ontology for the product in question
- Then **b** retrieves the relevant experience from the case base reasoning system

[10] Agents may enter and leave the marketplace at anytime and up-to-date information should be maintained in the buyer-seller list to find the supply demand ratio at any point in time.

- Based on its attitude, public price range, supply-demand ratio from list of Buyers&Sellers, and the relevant experience, **b** computes its own price range as a complex function (will be discussed in the following sections)
- If the actual buyer's maximum is less than **b**'s calculated maximum, then actual buyer's maximum becomes **b**'s maximum
- **b** gets the list of sellers who are selling the same product in the marketplace
- For each relevant sellers, **b** creates a buyer clone **bc** and sends a message to the seller requesting for negotiation
- Seller creates a seller clone **sc** for the respective **bc**
- Negotiation starts between each pair of **bc** and **sc**
- **bc** starts bidding with the minimum price and the **sc** starts counteroffer with its maximum price (here the agents price range are hidden)
- A deal is reached when the **bc**'s maximum reaches **sc**'s minimum
- No deal is made if the **bc**'s maximum is less than the **sc**'s minimum
- Now both the sellers and buyers experiences are saved to the case base for future use.

The process we described above becomes more complicated for buyers and sellers who are already negotiating in a dynamic marketplace. As the buyer agent (above) enters the marketplace, demand for the product increases. At this point in time, only the last buyer who just entered the marketplace knows the actual supply demand ratio for that product. The agents who were already in the market would not have the correct information of the product's supply-demand ratio anymore. [37] doesn't address this problem. In fact, it acts like a task oriented conventional system by negotiating for the product with inaccurate information without giving any consideration to the changes that occurred in the environment. This may lead to a variety of problems.

In a scenario where a buyer agent enters the marketplace when there are other buyer and seller agents already negotiating for the same product, the newly arrived buyer agent will calculate the supply-demand ratio lower than the other buyer agents. In this situation, the

last entered buyer agent will calculate its maximum price slightly higher than the other buyer agents[11]. Since the maximum price of the last buyer agent is higher, it has higher probability of making a better deal than other buyer agents. One could imagine other scenarios. For instance, when a seller agent enters the marketplace it may calculate the minimum price it is willing to accept lower than the other seller agents. This may lead the last entered seller agent to sell the product faster than other seller agents who are selling the same product.

An e-commerce system can be affected by many other external factors. A company's reputation or the quality of the product may play a major role in the general price range of a product. An external factor such as change in general price range of a product should be appropriately validated in the system so that the agents can be competitive during the negotiation process. Whenever there is a change for a product, internally or externally as in supply-demand ratio or in general price range, the e-commerce system should notify all the relevant participants in the system to avoid any inconsistencies among them.

An overview of the above process, as a buyer agent enters the marketplace when there are other buyers and sellers already negotiating for the same product in that system is described bellow:

- A buyer accesses the marketplace's interface and provides the information to create a buyer agent **b** with the product name, his/her attitude, warranty and maximum and minimum price for the product
- Environment Watcher, the central controller of the system, asks the list of Buyers&Sellers to create, register the agent and to return all the relevant buyers and sellers who are in the market for the same product
- List of Buyers&Sellers registers the agent **b** in its list of buyers and sellers
- **b** retrieves the publicly available price range from the Ontology for the product in question

[11] Provided that the attitudes of the other agents are similar and even if its not it is not fair for the last agent.

- Then **b** retrieves the relevant experience from the Case Base Reasoning system
- Based on its attitude, public price range, supply demand ratio and the relevant experience, **b** computes its own price range as a complex function (will be discussed in the following sections)
- If the actual buyer's maximum is less than **b**'s calculated maximum, then actual buyer's maximum becomes **b**'s maximum
- List of buyers and sellers returns Environment Watcher with buyers and sellers who are negotiating for the same product as **b**
- Environment Watcher sends the new supply-demand ratio to each buyer and seller agents who are negotiating for the same product in the marketplace
- All the buyers and sellers who's clones are still negotiating for the product recalculate their price range and pass their new maximum and minimum price range to their negotiating clones
- Clones continue to negotiate for the product without any interruption in the negotiation process with the new maximum and minimum price range
- Now **b** gets the list of sellers who are selling the same product in the market
- For each relevant sellers, **b** creates a buyer clone **bc** and sends a message to the seller requesting for negotiation
- Each seller creates a seller clone **sc** for the respective **bc**
- Negotiation starts between each pair of **bc** and **sc**
- **bc** starts bidding with the minimum price and **sc** starts counteroffer with its maximum price (here the agents price range are hidden)
- A deal is reached when the **bc**'s maximum reaches **sc**'s minimum
- No deal is made if the **bc**'s maximum is less than the **sc**'s minimum
- Now both the sellers and buyers experiences are saved to the case base for future use.

Moreover, when agents leave the market place, a similar procedure is followed to ensure that all negotiating agents have the up-to-date information about the product's supply-demand ratio in the marketplace.

At any given time, an external factor such as public price range may change outside the e-commerce system for a product. Any external changes that might potentially affect the agents' negotiation process should immediately be updated in the system so that the e-commerce system will be consistent with the dynamic and uncertain real world. We consider the change in general price range for a product as one of the external factors that affects the agents' negotiation process in this dynamic negotiation model. The following process is executed when there is a change in general price range outside the e-commerce system:

- Administrator of the system will notify the e-commerce environment that there is a change in price range for product X through the Environment Watcher
- Environment Watcher will update the Ontology where the domain specific information of the product X is kept
- Environment Watcher will ask the list of Buyers&Sellers to return all the relevant buyers and sellers who are negotiating for the product X, if any, in the marketplace
- List of Buyers&Sellers will return Environment Watcher with buyers and sellers who are negotiating for the product X
- Environment Watcher will send the new minimum and maximum general price range to each buyers and sellers who are negotiating for the product X
- All the buyers and sellers who's clones are still negotiating for the product X will recalculate their price range and pass their new maximum and minimum price range to their negotiating clones
- Clones will continue to negotiate for the product without any interruption in the negotiation process with the new maximum and minimum price range

The processes described above subsume the dynamic negotiation process in this model.

3.5 Definitions used in [37 & 43]

3.5.1 Agent's Attitude

An agent's attitude is a hidden mental state comprised of triple (IOTime, IOPrice, Commit) representing the importance of time, the importance of price, and the commitment level of an agent.

Urgency and commitment reflects the desperateness to purchase or sell the product. Importance of Price determines the maximum a user can spend for the purchase. IOTime, IOPrice and Commit take on values in the open interval of [0, 1]. For instance,
An agent with attitude of (1.0, 0.2, 0.8) represents,

- Time is priority
- Price is not important
- Commitment level is rather high

Note that these agents' mental attitude attributes are hidden from opponent agents.

3.5.2 Public Price Range (PPR)

All agents in the marketplace are assumed to have access to a product price range, which is the general public price range for a product that can be obtained from the Ontology. Ontology[12] returns a public price range of min and max for the given product.

$$PPR(prod) = [pmin, pmax]$$

3.5.3 Supply Demand Ratio

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

$$SDR = |S| / |B|$$

3.5.4 Negotiation

[12] Ontology, as in [Saba & Sathi, 2001], is a hierarchy of concepts. It supplies the domain specific knowledge for the agents.

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

DONE $^{+}$	: Negotiation completed Successfully
DONE $^{-}$	: Negotiation completed Unsuccessfully
DONE ^{0}	: Negotiation is still in progress

A negotiation record ({ (offer, counteroffer) }, DONE ^{x}) is an ordered list that contains, offers and counteroffers and an outcome of the negotiation.

3.5.5 Agent Experience

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

PCat	PN	Warr	PPR	Price	Attitude	SDR	NegRec	Res
Helec	36"TV	2	1200-2000	1500	1.0,0.2,0.8	2	1250/1800/1400/ 1600/1500/1500	1

The above experience represents a buyer agent's experience in buying a 36" TV, when the supply demand was 2 is 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 six exchanges of offers and counter offers. This experience could be used in the future when buying similar products.

3.5.6 Agent's Price Range (APR)

An agent's price range is a function of agent's attitude, public price range, supply demand ratio, warranty and prior experiences.

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

$$= [<pmin + (t)(pmax\text{-}pmin)/\Psi >,$$

$$< pmax - (pmax)(p)(1\text{-}c)/\Psi + pmax(1\text{-}SDR)\epsilon + pmax(War_{cur} - War_{exp})\gamma>]$$

where, SDR – Supply Demand Ratio in the current market,

War_{cur} - Warranty in current situation,

War_{exp} - Warranty in experience,

pmin and pmax are minimum and maximum price from PPR, and

Ψ, ϵ and γ are constraints which vary based on the domain.

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

$$= [< pmin + ((pmin)(p)(1\text{-}c)/\Psi) + pmin(1\text{-}SDR)\epsilon + pmin(War_{cur} - War_{exp})\gamma >,$$

$$< pmax - (t)(pmax\text{-}pmin)/\ \Psi >]$$

3.6 Dynamic 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), SDR, Warr)
- **s** computes its price range: $[spr_{max}, spr_{min}] \leftarrow APR^{seller}$ ([pmin, pmax], (st, sp, sc), SDR, Warr)
- Buyers and Sellers who are already in the marketplace and negotiating for the same product are notified of the change in the environment by the Environment Watcher
- Agents which are already in the environment recalculate their price ranges with the new SDR and pass the new minimum and maximum to their clones
- **b** hides its bpr_{max} and starts its bidding with $b_{bid} \leftarrow bpr_{min}$
- **s** hides its spr_{min} and starts its bidding with $s_{bid} \leftarrow spr_{max}$
- With each successive offer and counteroffer, buyers and sellers update their respective biddings as follows: $b_{bid} \leftarrow (b_{bid} + \alpha)$ and $s_{bid} \leftarrow (s_{bid} - \beta)$

- α and β are the buyer's step increment and the seller's step decrement, respectively. α and β are calculated using their similar buying or selling experiences and present market conditions.
- A negotiation is always in one of the following states:
    - $DONE^{+}$ if $b_{bid} \geq s_{bid}$
    - $DONE^{-}$ if $bpr_{max} < spr_{min}$ (these are hidden from each other)
    - $DONE^{0}$ if $(b_{bid} < s_{bid}) \wedge (bpr_{max} \leq spr_{min})$

Moreover, a buyer may concurrently negotiate with as many sellers and a seller may also concurrently negotiate with as many buyers in the marketplace using their buyer and seller clones. The *best seller* for a buyer would be the seller with lowest agreed price among the successful negotiations. Similarly, the *best buyer* for a seller would be the buyer with highest agreed price among the successful negotiations. When all the clones of a buyer finish negotiation with their respective seller clones, buyer will request the *best seller* to sell the product. However, a seller (seller's clones) may not be done negotiating with all other buyers in the marketplace. It is important for a seller agent to wait till it finishes negotiation with all the buyers in the marketplace since there may be other buyers who might be willing to pay a higher price for the product.

In a scenario where a seller agent is not done negotiating with all the buyers and yet a buyer wants to buy the product, the seller agent will ask the buyer to wait. The buyer however will query the *best seller* to sell the product again and again till he gets a result of either 'No Deal' or 'Done Deal' from the seller. When the seller finishes negotiation with all the buyers, it will select the *best buyer* to sell the product. Again, the seller may give a wait note to any buyers who are asking it to sell the product, if none of those buyer agents are *best buyer* for this seller. Otherwise, the seller will sell the product to the *best buyer*, notify all the buyers with whom it had successful negotiations and then remove itself from the marketplace.

There may be another situation where a seller may wait for the *best buyer* by giving wait message to any other buyers who are requesting to buy the product. If that *best buyer* has another seller, who is offering lower price than this seller, it will finish the deal with the other seller. Once the buyer finishes the deal, it will notify all the sellers, including waiting seller that it doesn't need the product anymore. At this point, the seller agent will remove the *best buyer* and choose the next *best buyer* agent from successful negotiation list. When the *best buyer* asks for the product again, seller will sell the product, notify all the buyers with whom it had successful negotiations and remove itself from the marketplace.

3.7 Competitiveness and Cooperativeness of Dynamically Negotiating Agents

Dynamically negotiating agents in this model shares the same views and ideas described in section 2.6. Our model creates dynamically negotiating agents with competitive and cooperative settings in a single negotiating model. Buying and selling agents have competing interests because they both want to gain more profit (refer to section 2.6). Yet, they indirectly cooperate with each other when their hidden mental attitude agrees (refer to section 2.6). In this sense, we have both competitive and cooperative dynamic negotiation in our framework.

3.8 Learning from Experience in the framework

The agents in this model, as in [37, 38,43] learn from their experience over a period of time and they reason accordingly during negotiation. Successful and unsuccessful negotiations are stored in the case based reasoning system for future use since learning can occur from both successful and failed negotiations. Specifically, prior experiences are used to adjust agents' attitudes slightly and calculate price ranges, bid increments and bid decrements for offers and counteroffers of buyer and seller agents. A relevant negotiation experience is found from the Case Based Reasoning system and it is used to slightly adjust an agent's mental attitude. The reason behind this process is to bias the agents

towards successful negotiations. However, the attitude adjustments are done within certain range to ensure that the agents do not deviate from their needs and goals.

3.8.1 Matching Cases

When searching for a relevant case a perfect mach cannot be expected. Process of finding a relevant case starts with finding all the experience records from the Case based Reasoning system for a product. This may contain both successful and unsuccessful negotiations. When searching for a 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,

The range similarity falls between [0, 1.0] intervals. The attributes to be matched include product, attitude, and public price range of the product. The most relevant case to the current scenario is the highest match found by the above function from the Case Based Reasoning system for the product in question.

3.8.1.1 Product similarity

The product similarity consists of both conceptual similarity and price similarity. The reason behind considering both price similarity and conceptual similarity can be explained as follows:

"When buying a scanner, one might recall their experience of buying a printer. In this case conceptual similarity seems to be sufficient. However, one may not recall buying a computer monitor when buying a computer mouse, although they are conceptually similar. Clearly, here the price range also plays an important role in finding the product similarity."

Therefore, similarity between two products can be a function of both price similarity and conceptual similarity as follows.

*ProductSimilarity PS (r1, pr2, prod1, prod2) = (PrS(pr1, pr2)) * (CS(prod1,prod2))*

where, PrS and CS are defined as follows

3.8.1.1.1 Price Similarity (PrS)

The following function is used to find the price similarity.

$$\text{PriceSimilarity}(\text{pr1}, \text{pr2}) = \left(1 - \frac{\text{abs}(\text{pr1} - \text{pr2})}{\text{max}(\text{pr1}, \text{pr20}}\right)$$

Where Pr1 and Pr2 are the average price range for product1 and product2 respectively.

3.8.1.1.2 Conceptual Similarity (CS)

The conceptual similarity for two products is measured from the Ontology. Agents in the virtual market place have access to Ontology where the domain specific information of a product is kept. Using the notion of semantic distance in a semantic network, the simple measure of conceptual similarity between two products may be measured as follows:

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 instance, assume that a monitor and printer are Computer products as shown in figure 3.2 and the least upper bound between them is 1.

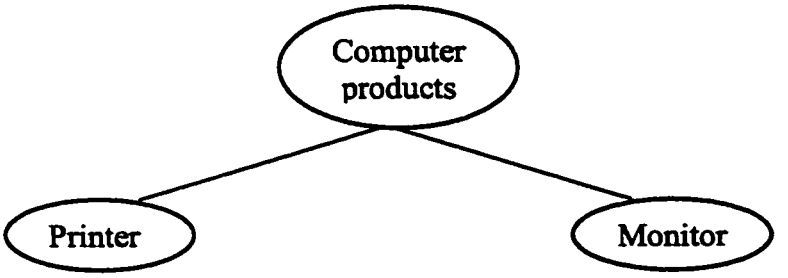

Figure 3.2: Conceptual Similarity

Then the distance from printer to Computer products is 1 and the distance from monitor to Computer products is also 1. Conceptual similarity between these two products are computed as follows:

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 (that the is printer and the monitor are very similar conceptually)

3.8.1.2 Attitude Similarity

Attitude of a buyer or a seller defines the buying or selling behavior of an agent. The similarity between the attitudes of two agents is computed as follows:

$$AS(<t_1,p_1,c_1>,<t_2,p_2,c_2>) = ((1 - |t_1 - t_2|)+(1 - |p_1 - p_2|)+(1 - |c_1 - c_2|)) / 3$$

where $<t_1,p_1,c_1>,<t_2,p_2,c_2>$ are attitudes of agent1 and agent2 respectively.

3.8.1.3 Public Price Range (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:

$$RS([\min 1, \max 1],[\min 2, \max 2]) = \left(1 - \frac{abs(\min 1 - \min 2)}{\max(\min 1, \min 2)}\right)\left(1 - \frac{abs(\max 1 - \max 2)}{\max(\max 1 + \max 2)}\right)$$

where min1 and max1 are the public price range of product1 and min2 and max2 are the public price range of product2.

3.8.2 Adjusting Attitude

The attitude of an agent is adjusted based on previous experience. The experience we use here is the best-matched case received from the function in section 3.8.1. Finding optimal attitude is an important part of the learning process, as it leads to more successful negotiations. While calculating optimal attitude for a given situation, agents should not

deviate completely from their clients needs. Following function is used to adjust the attitude of a buyer agent using both successful and unsuccessful experiences for a negotiation:

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

$t_{cur} = t_{exp}$ if outcome(Neg) = DONE$^+$ & abs($t_{exp} - t_{cur}$) ≤ 0.1

$t_{cur} = t_{cur} + 0.1$ if outcome(Neg) = DONE$^+$ & $t_{exp} > t_{cur}$

$t_{cur} = t_{cur} - 0.1$ if outcome(Neg) = DONE$^+$ & $t_{exp} < t_{cur}$

where t_{exp} is the Importance of Time attributes in the best relevant experience.

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

$p_{cur} = p_{exp}$ if outcome(Neg) = DONE$^+$ & abs($p_{exp} - p_{cur}$) ≤ 0.1

$p_{cur} = p_{cur-0.1}$ if outcome(Neg) = DONE$^+$ & $p_{exp} < p_{cur}$

$p_{cur} = p_{cur+0.1}$ if outcome(Neg) = DONE$^+$ & $p_{exp} > p_{cur}$

where p_{exp} is the Importance of Price attributes in the best relevant experience.

Similarly, commitment could also be adjusted based on previous experience. It can be observed from the above functions that a threshold of 0.1 bounds the difference between adjusted attitude and initial attitude. This is to control the agents from deviating from their clients' goals.

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

$t_{cur} = t_{cur} + 0.1$ if outcome(Neg) = DONE$^-$ & $t_{exp} > t_{cur}$

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

$p_{cur} = p_{cur} - 0.1$ if outcome (Neg) = DONE$^-$ & $p_{exp} < p_{cur}$

In contrast to what we observed in using successful experience, the agents tend to behave completely different in using negative experiences.

Therefore, agents in our model adjust their attitude to find an optimal attitude using previous experiences. Then they calculate their price range to reflect the experience and their adjusted attitude.

3.8.3 Bid Increment

The following function was suggested to calculate bid increments.

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

where Att' is the adjusted attitude of an agent, p' = 0.1, if p' = 0.0 and APR = [APR_{min}, APR_{max}]

This function assumes an average of 20% increment and it also takes into account of Importance of time and Importance of Price while calculating the bid increments. After calculating the 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 exceeds the seller's AprMin.

3.8.4 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 a new experience is matched with the one in the Case Base. When searching for updating or adding a new experience, a match between two cases is done as follows:

$$\text{Match}(c1, c2) = \frac{1}{4} \left(\begin{array}{l} w1 * \text{PS}(\text{prod}(c1), \text{prod}(c2)) + w2 * \text{AS}(\text{att}(c1), \text{att}(c2)) + \\ w3 * \text{RS}(\text{ppr}(c1), \text{ppr}(c2)) + w4 * \text{NS}(\text{neg}(c1), \text{neg}(c2)) \end{array} \right)$$

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

3.8.4.1 Negotiation similarity

Negotiation similarity is measured in terms of number of offer exchanges and negotiation result. Number of exchanges can be looked as the time taken for the negotiation. Equal priority is given to both result of the negotiation and number of exchanges occurred in calculating negotiation similarity.

$$NS(\langle L1, outcome1\rangle, \langle L2, outcome2\rangle)$$

$$\begin{cases} 1-((abs(N1-N2)/\max(N1,N2))*0.5+0.5) & \text{if outcome1} <> \text{outcome2} \\ 1-abs(N1-N2)/\max(N1,N2) & \text{otherwise} \end{cases}$$

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

3.8.4.2 Updating Case Base

Efficiency of the system depends on how the agent experiences are stored in the case base system. One has to be careful in storing new cases, so that case base is not populated exponentially. Method of storage should take into consideration both conceptual and computational gain. Conceptual gain is when the agents refine their memory by adding new cases to what they already know. Computational gain is that the system would be faster as the retrieval of cases is reduced.

When matching new experience with the old ones in the Case Base, if a strong match is not found then the new experience is saved as a novel experience in the Case Based system. Otherwise, new experience is merged with the strong matched old experience as follows:

Merge(<p1,pn1,ppr1,wr1,pr1,att1,sdr1,neg1>,< p2,pn2,ppr2,wr2,pr2,att2,sdr2,neg2>)

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

Merging of two cases occurs only when both products are same and the price range is similar. When merging, price average, attitude and supply demand ratio of the two cases are computed. Negotiation that took the minimum offer exchanges is also selected. Then the above attributes are replaced in the old case to refine the old experience.

3.9 Conclusion

In this chapter, we proposed a framework for automated dynamic negotiation in e-commerce based on the principle of BDI model of agency. The buying and selling agents in this framework as in [37] autonomously negotiate on behalf of their clients with a given mental attitude and previous similar negotiating experiences. Furthermore, we presented the effect of dynamicity inside and outside the marketplace that could affect the negotiation process of the buying and selling agents in an e-commerce environment and illustrated how to handle this problem with the use of a decision making model, BDI model of agency.

The agents in this framework are goal oriented and they negotiate competitively and cooperatively for their own benefits [36, 38]. Buyer and seller agents in this framework negotiate simultaneously with as many opponents in the marketplace to find the best deal among all participants. The framework proposed in this chapter attempts to model the human negotiating behavior into buying and selling agents in an e-commerce environment even when there are uncertainties in the marketplace.

Chapter 4

Design and Implementation Details

The model discussed in the previous Chapter attempts to automate dynamic negotiation in an e-commerce system. This Chapter presents design and implementation details of the ideas discussed in the last Chapter. The framework is implemented in Java programming language with approximately three thousand lines.

4.1 E-Commerce Environment

The e-commerce environment consists of the following parts:

- an Environment Watcher which acts as the controller of the system
- list of Buyers&sellers who are currently in the marketplace
- Ontology with the domain specific information on products and
- Case Based Reasoning System with prior buying-selling experiences

A buyer or a seller provides the following information to create an agent in the system:

- name for the agent
- name of the product
- type of the agent (buyer or seller)
- attitude towards buying or selling the product
- warranty expected (buyer) or provided (seller)
- maximum and minimum price range he/she is willing to go

Users create agents at their own will and send them to the marketplace for negotiation. Creation or disposal of an agent is always carried out through the Environment Watcher in this dynamic e-commerce system. Environment Watcher is also responsible for notifying negotiating agents of any changes that occurs inside or outside the marketplace. The list of Buyers&Sellers keeps the up-to-date references of all the buying and selling agents. Ontology is made public to all the agents to access the product domain-specific

information. Case Based Reasoning system allows buyer agents to retrieve buyers' prior experiences and seller agents to retrieve sellers' prior experiences separately.

4.2 Object Diagram of the Framework

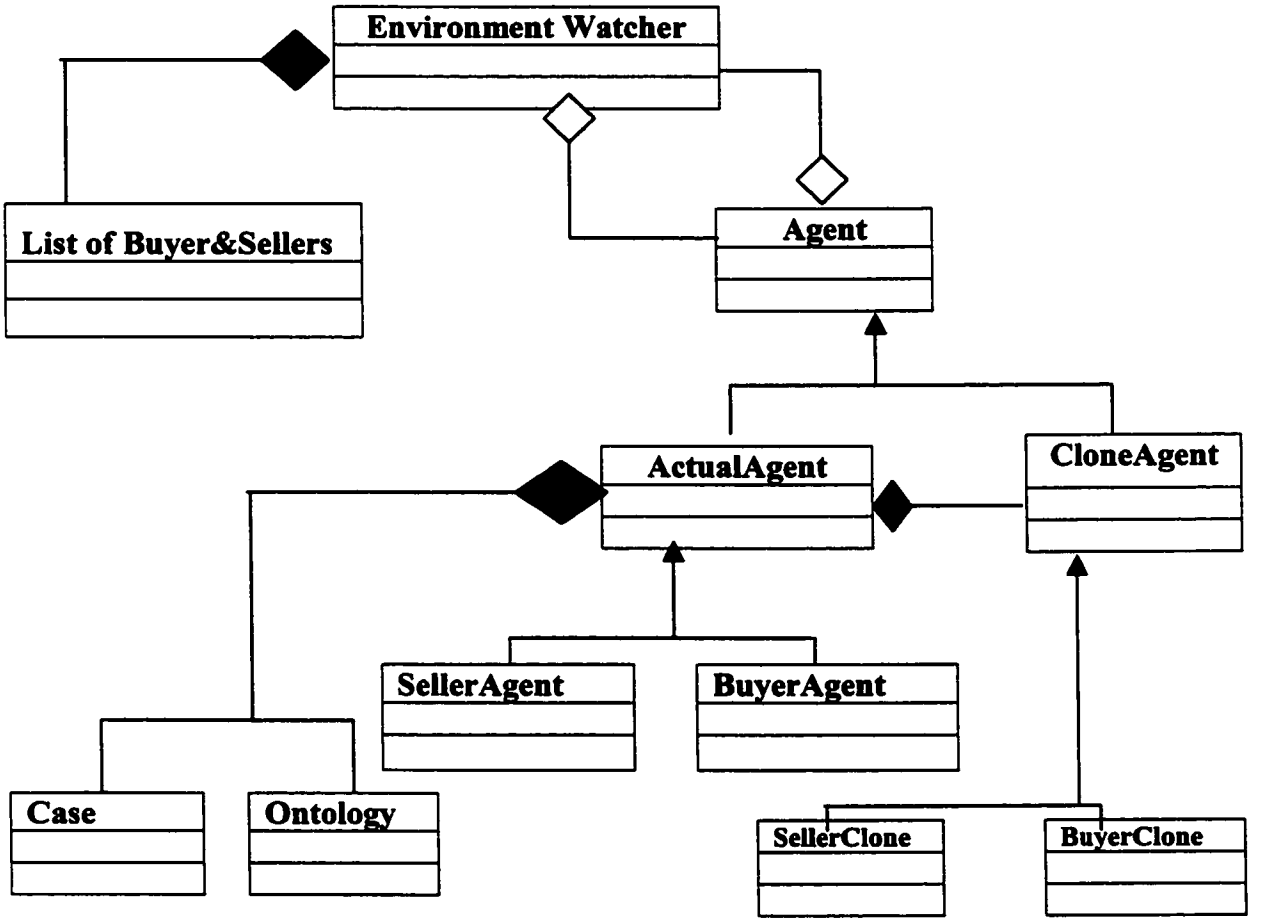

Figure 4.1: Object Diagram for the framework

Major Objects in this model include EnvironmentWatcher, BuyerSellerList, Agent, ActualAgent, CloneAgent, Case and Ontology. ActualAgent and CloneAgent are an extension of type Agent. BuyerAgent and SellerAgent are an extension of ActualAgent and BuyerClone and SellerClone are an extension of CloneAgent. Basically, all the agents in this model are an extension of type Agent as shown in the Figure 4.1. EnvironmentWatcher interacts with BuyerSellerList, BuyerAgent and SellerAgent in two

situations. One situation is when there is a need to create or dispose an agent and the other is when there is a change in Public Price Range for a product. Furthermore, ActualAgents interact with Case and Ontology to compute their respective maximum and minimum prices. They also interact with EnvironmentWatcher when a deal is done successfully or unsuccessfully in order to remove themselves from the marketplace. BuyerAgents and SellerAgents interact with each other in order to buy or sell the product. Interaction between BuyerAgents and their BuyerClones are mutual as there is exchange of messages during the process of negotiation. SellerAgents also interact with their SellerClones for the same reason. Moreover, there is a heavy interaction between the BuyerClones and SellerClones when Clones they try to find the best deal in the marketplace.

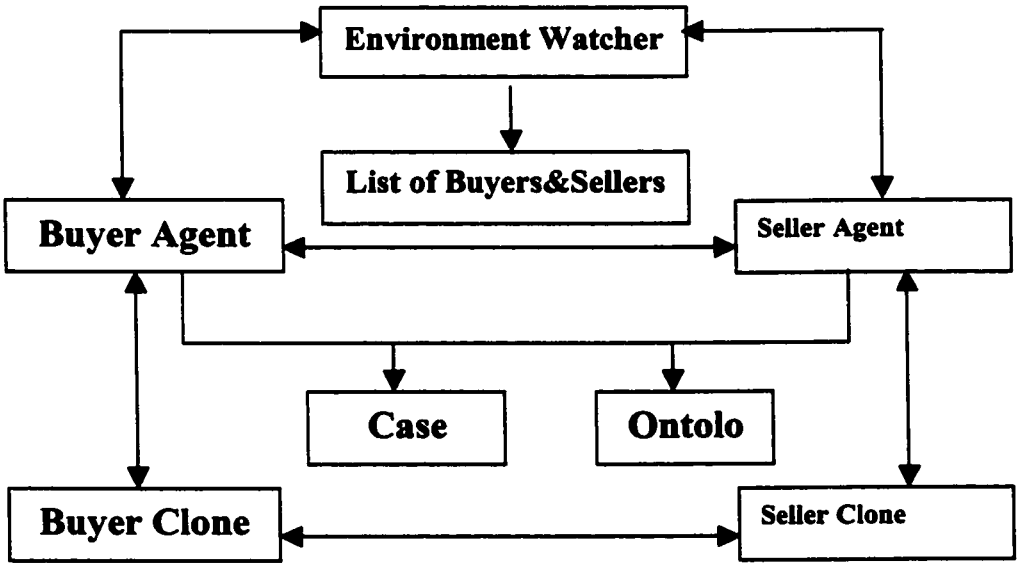

Figure 4.2: Object interactions in the framework

4.3 Creating Agents and Clones

4.3.1 Creating an Agent

Agents in this model learn from prior experiences by adopting relevant experiences from the Case Base System. They also use the domain specific information from Ontology when calculating their maximum and minimum acceptable prices. Agents on creation

retrieve similar product information from the Ontology and they also retrieve relevant experience from Case Based Reasoning System.

4.3.1.1 Retrieving Similar Products from Ontology

Ontology serves two purposes in this framework as mentioned in the previous Chapter.

- Agents retrieve public price range from the Ontology and
- When 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 tables 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 figure 4.3.

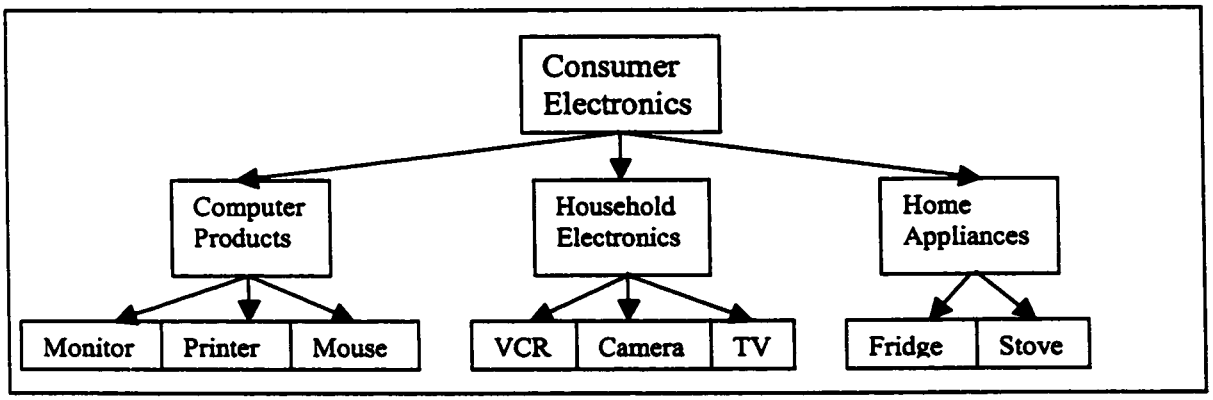

Figure 4.3: Ontology for Consumer Electronics

Conceptual similarity of two products ranges between intervals of <0.0, 1.0>. The conceptual similarity between two products under the same concept takes the value of 1.0. For example, Monitor, Printer and Mouse are under the same concept, i.e. Computer Products. Therefore, conceptual similarity of Monitor and Printer or Monitor and Mouse will take the value of 1.0. When comparing Monitor and TV, the conceptual similarity between them is 0.5 because Monitor and TV are under two different concepts and those

concepts are under the same concept, i.e. Consumer Electronics. The Ontology that is assumed in figure 4.3 is represented in the form of database tables as shown in the table bellow:

Prod Category	Prod Name	Min Price	Max Price
CompProd	Monitor	300	400
CompProd	Printer	200	350
CompProd	Mouse	10	15
HomeElec	Camera	500	900
HomeElec	TV	300	500
HomeElec	VCR	100	200
HomeAppl	Fridge	500	750
HomeAppl	Stove	350	500

Product Category 1	Product Category 2	Value
CompProd	CompProd	1
CompProd	HomeElec	0.5
CompProd	HomeAppl	0.5
HomeElec	CompProd	0.5
HomeElec	HomeElec	1
HomeElec	HomeAppl	0.5
HomeAppl	CompProd	0.5
HomeAppl	HomeElec	0.5
HomeAppl	HomeAppl	1

Table 4.1: Database tables representing the Conceptual hierarchy

The table on the left represents the public price range. Minimum and Maximum price of a product is retrieved using the key ProdName. Table on the right represents the conceptual similarity between products considered. Conceptual similarity value is retrieved using the combination of ProductCategory1 and ProductCategory2 as key.

4.3.1.2 Retrieving Relevant Experience

Buying and selling agents explore the Case Base on entry into the marketplace to retrieve relevant experiences. As discussed in the previous Chapter, this framework uses Case Base Reasoning techniques to learn from past experiences. Each cases in the Case Base represents a unique experience. They contain specific problem situation, solution to problem under that situation and result of applying that solution to problem in that situation.

4.3.1.2.1 Representing Cases (Experience)

A buyer's negotiation experience with the result of negotiation is represented in a database table as follows:

Pcat	Pname	War	PPR	Price	Attitude	SDR	NegRec	Res
EleEnt	36"TV	2	1200-2000	1500	1.0,0.2,0.8	2	1250/1800/1400/1600/1500	1
CmpPro	Monitor	1	200-300	260	0.5,0.8,0.8	0.5	200/310/220/290/260	0

Table 4.2: Table representing buyers' experiences

Attribute NegRec in the above table represents the offers and counteroffers between the buyer clone and the seller clone. Res attribute represents the outcome of the negotiation. Result '1' implies successful negotiation and '0' implies unsuccessful negotiation.

4.3.1.2.2 Choosing Best Relevant Experience

The function presented in section 3.8.1 of chapter 3 is directly implemented to find the relevant matching experience. We used a minimum threshold of 0.7 to choose the best relevant case.

4.3.2 Creating Clones

A buyer agent is responsible for creating buyer clones and a seller agent is responsible for creating seller clones. Initially, buyer agent creates a buyer clone for each seller present in the market. When a buyer clone is created, it requests the corresponding seller for negotiation. Seller agent in turn uses the buyer clone's information to find who is the buyer agent and then creates a seller clone for that buyer clone; finally, seller clone returns its reference to the buyer clone for further communication during the negotiation.

Clones in this model propose and counter-propose to their opponent clones to find a successful deal. On creation, a clone receives negotiable price range from its parent agent. Using the function suggested in 3.8.3 in Chapter 3, clones calculate their bid increment or decrement prices. Even though clones actually do the negotiation, clones'

parent agents make the final purchasing decision in this model. Although clones are equipped with the negotiation strategies, parent agents of this model hold the final decision making capabilities.

4.4 Disposing Agents and Clones

In order for the e-commerce system to be efficient and fast, all the agents and their clones should be disposed once the negotiation process is successfully or unsuccessfully ended.

4.4.1 Disposal of an Agent

A buyer or seller agent's disposal process starts when an agent requests the Environment Watcher to remove itself from the marketplace and it ends when the Environment Watcher removes the agent from the list of Buyers&Sellers. However, before an agent requests the Environment Watcher to remove itself from the marketplace, it updates its experience to the Case Base System.

4.4.1.1 Updating a Relevant Experience

It is necessary to update the Case Base after each negotiation so that the experience could be used in future endeavors of these agents. As discussed in section 3.8.4 of Chapter 3, the new solution is evaluated and matched with the existing cases in Case Base. If the new solution matches to a certain threshold with any existing cases in Case Base, then that case in the Case Base is updated using the function presented in section 3.8.4.2 of chapter 3. Otherwise the new solution is stored as a novel experience.

4.4.2 Disposal of Clones

Each clone's negotiation with its opponent must end successfully or unsuccessfully. When a negotiation is over, information to contact the opponent agent and information regarding the negotiation process is passed to the parent agent. Parent agent saves the clone's negotiation information for future use and then disposes the corresponding clone. During this process, parent agent checks for negotiating clones. If there are no more clones negotiating for the product, parent agent finds the best buyer or seller out of all

successful negotiations to make the final deal. The procedures discussed so far subsume the dynamic negotiation process of this framework.

4.5 Implementing Functionalities of the EnvironmentWatcher

The functionalities of the EnvironmentWatcher discussed in section 4.1 are implemented in class EnvironmentWatcher. EnvironmentWatcher has methods *AddBuyerSeller*, *RemoveBuyerSeller* and *PublicPriceRangeChangeForProduct* for registering an agent to the environment, disposing an agent from the environment and changing the public price range of a product in the Ontology. When *AddBuyerSeller* or *RemoveBuyerSeller* method is invoked, it dynamically finds the supply-demand ratio and also the buyers and sellers in the marketplace for the product in question using class BuyerSellerList. When there is at least one buyer and seller negotiating for a product and when the change in supply-demand ratio is higher than 0.1, AddBuyerSeller or RemoveBuyerSeller notifies all buyers and sellers who are negotiating for that product about the change that occurred in the marketplace. Method *PublicPriceRangeChangeForProduct* is invoked whenever the public price range for a product changes. It uses the class BuyerSellerList to dynamically find the buyers and sellers negotiating for a product. Then it updates the new public price range in the Ontology and notifies all relevant agents about the change in public price range.

4.6 Implementing Functionalities of the BuyerSellerList

The functionalities of the dynamic list of Buyers&Sellers are implemented in class BuyerSellerList. BuyerSellerList has methods *AddBuyerSeller*, *RemoveBuyerSeller* and *WhoAreTheBuyersSellers* for adding an agent, removing an agent and collecting the references of all relevant[13] agents from the list of Buyers&Sellers. Before adding an agent to the list, *AddBuyerSeller* method checks to see if the agent name already exists in the list. If it exists, it sends message 'Name Exists' to EnvironmentWatcher. Otherwise, the agent is added to the list of Buyers&Sellers and references of all agents who are

[13] Relevant agents' references for the product in question.

negotiating for the same product is returned to the EnvironmentWatcher. Before removing an agent from the list, *RemoveBuyerSeller* method checks to see if the agent name exists in the list. If it exists, the agent is removed from the list of Buyers&Sellers. Otherwise, it sends a 'None' message to EnvironmentWatcher. When method *WhoAreTheBuyersSellers* is invoked, it returns references of all the agents who are involved in buying and selling the same product to the EnvironmentWatcher.

4.7 Implementing Functionalities of the Ontology

The functionality of the Ontology discussed in section 4.3.1.1 is implemented in class Ontology. Table 'RANGE' and 'SIMVALUE' in Oracle database represent Domain Specific Knowledge and Conceptual Similarity, respectively. Public price range for a product is retrieved from table 'RANGE' using *GetProdDetails* method. Method *ProductSimilarity* in class Ontology is used to find the similarity between two products. Connection to Oracle database is established with JDBC Bridge.

4.8 Implementing Functionalities of the Case Base

Agent experiences are modeled in two separate tables as discussed in section 4.3.1.2.1. Table BUYEREXP in database represents buyers' experiences and table SELLEREXP represents sellers' experiences. Cases in both tables are indexed based on their product name. Class Case returns best relevant record from the Case Base for a given situation. Method *GetBestCase* of class Case returns the best relevant experience record as a string. It uses method *ProductSimilarity* to find the similar products from class Ontology. Method *AttitudeSimilarity* and *RangeSimilarity* are used to find the similarity between two attitudes and price ranges respectively. Method *UpdateOrAddNegotiation* compares the similarity between the current case and the cases in the Case Base with the additional similarity check, *NegotiationSimilarity*. If the match threshold is above 0.7 it updates the exiting case. Otherwise it stores the case as new case as discussed in section 3.8.4 of Chapter 3.

4.9 Implementing the Functionalities of the Agents

Class Agent is an abstract class that defines the most common characteristics and functionalities of all the agents in the system (*refer to figure 4.1*). Although, all agents are of type Agent, ActualAgent and CloneAgent differ substantially because of their subclass' functionalities. Therefore, two types of constructors are provided in class Agent, one to create ActualAgent and the other to create CloneAgent. ActualAgent and CloneAgent are also abstract classes since the common characteristics and functionalities of BuyerAgent and SellerAgent, BuyerClone and SellerClone are implemented in ActualAgent and CloneAgent, respectively. The relevant experiences for both BuyerAgent and SellerAgent are retrieved and the mental attitudes are adjusted using the method *AdjustAttitude* in ActualAgent.

4.9.1 Buyer Agent

Class BuyerAgent uses method *FindMaxMinPrice* to find the price range. When EnvironmentWatcher notifies arrival of new seller agents in the marketplace and the change in supply-demand ratio, *Notify* method invokes *CreateClone* method to create new BuyerClones. Now *CreateClone* method sends message to the new SellerAgent and receives a SellerClone reference for further negotiation. Method *FindMaxMinPrice* is invoked at this point to recalculate the price range of the BuyerAgent. Once the new price range is found, it is broadcasted to all the BuyerClones of this BuyerAgent. When EnvironmentWatcher notifies change in public price range, another *Notify* method is invoked in BuyerAgent. This *Notify* method in turn invokes *FindMaxMinPrice* again to recalculate the price range. At this point, new price range gets broadcasted to all the clones of this BuyerAgent. When all the clones of a BuyerAgent finish negotiation successfully or unsuccessfully, method *WantToBuy* is invoked to contact the best seller agent. Method *WantToBuy* finds the best seller by invoking *SuccessfulDealByClone* method. If a SellerAgent is not done negotiating with all the BuyerAgents in the marketplace, it will send a message 'wait' to any BuyerAgent requesting to buy the product. When a 'wait' message is received in *WantToBuy* method of a BuyerAgent,

method *WantToBuy* invokes *CallWait* method. *CallWait* method in turn periodically requests SellerAgent to sell the product. If the BuyerAgent looses a deal with its best seller, method *WantToBuy* finds the next best seller by invoking method *SuccessfulDealByClone* again. Otherwise, when BuyerAgent finds a deal with a SellerAgent, method *SetStringForUpdation* is invoked to update the Case Base System. Once the Case Base is updated, BuyerAgent will send a reject message to all the other SellerAgents with whom it had successful negotiations and then it will send a message to EnvironmentWatcher to be removed from the marketplace.

4.9.2 Seller Agent

Class SellerAgent uses method *FindMaxMinPrice* to find the price range. When BuyerClone requests for negotiation, *SetBuyerClone* method in SellerAgent is invoked to create a new SellerClone. Method *SetBuyerClone* also saves the reference of the BuyerClone and returns newly created SellerClone's reference to the BuyerClone. When EnvironmentWatcher notifies arrival of new agents in the marketplace with the new supply-demand ratio or the change in public price range, *Notify* method of a SellerAgent invokes *FindMaxMinPrice* again to recalculate new price range. Once the new price range is found, it is broadcasted to all the SellerClones of this SellerAgent. Method *WantToBuy* of a SellerAgent may be periodically invoked by BuyerAgents. However, SellerAgent decides whom to sell the product only when all its clones are done negotiating with their corresponding BuyerClones. If a SellerAgent's clones are not done negotiating with corresponding BuyerClones, it will send a 'wait' message to any BuyerAgents who is requesting to buy the product. Method *WantToBuy* is also responsible for finding best seller and to finish the deal with BuyerAgent. When a deal is done, method *SetStringForUpdation* is invoked to update the Case Base System. Once the Case Base is updated, SellerAgent will reject all the BuyerAgents who are still requesting to buy the product and then it will send a message to EnvironmentWatcher to be removed from the marketplace.

4.9.3 Buyer Clones

The functionalities of a BuyerClone are finding bid increment, calculating offer and counter offer, recalculation of bid increment when there is a change in marketplace and notifying BuyerAgent when the negotiation is over. Method *BidIncrement* is used to calculate and recalculate the bid increment for the negotiation. Method *Offer* and *CounterOffer* are controlled by method *RunNegotiation* to appropriately offer and counteroffer bids to the seller clone. Method *CloneNegotiationAcceptReject* of BuyerClone is used to notify BuyerAgent any information about the negotiation process.

4.9.4 Seller Clones

Functionalities of a SellerClone are very similar to the functionalities of the BuyerClone. The functionalities include finding bid decrement, calculating offer and counter offer, recalculation of bid decrement when there is a change in marketplace and notifying SellerAgent when the negotiation is over. Method *BidDecrement* is used to calculate and recalculate the bid decrement for the negotiation. Method *Offer* and *CounterOffer* are controlled by method *RunNegotiation* to appropriately offer and counteroffer bids to the buyer clone. Method *CloneNegotiationAcceptReject* of SellerClone is used to notify SellerAgent any information regarding the negotiation process.

4.10 Challenges in Implementation

Controlling the flow of communication between agents and their clones as well as the communication between the clones is the toughest part of the implementation. Any agents in the marketplace need to keep track of their clones and opponents till the negotiation process is over. Moreover, the dynamicity that we adapted in this framework by allowing agents to enter and leave the marketplace at their own-will, increased considerable complexity to the application. Another challenge in creating this system appeared in the testing phase. Finding data that reflect the real world events and mental attitudes that replicate human buying behavior were another hardest part while testing this system.

Chapter 5

Evaluation of the Proposed Model

This Chapter discusses the evaluation phase of the proposed framework. The proposed research methodology in this work defines and formalizes the dynamic negotiation aspect for experienced agents with attitude in a virtual marketplace. First, we investigate and analyze the behavior of agents that support the dynamic negotiation in this framework. This analysis lays foundation for the experimental results. Second, we summarize the results presented in the previous model. Third section presents the need for adaptation of dynamic negotiation and evaluates [43]. Then we evaluate our model, taking into account of the dynamicity of an e-commerce environment. Finally, we present the computational issues involved in the proposed framework.

5.1 Behavior of agents in the model

The major issue of this research work is in representing dynamic negotiation capabilities into attitude and experienced based buying and selling e-commerce agents. The behavior of these agents is modeled based on mental attitude, previous buying-selling experiences and dynamic decision-making capabilities.

According to 43, human buying behavior heavily depends on attitude and experience. Further they believe importance of time, importance of price and commitment, which are the attributes of an agent's mental attitude, are very crucial in human buying behavior as they effect purchasing decisions of humans in real life. This is the rationality behind considering attitude when modeling agents in this framework.

The next important aspect presented in 43 is the use of previous experiences by the buying and selling agents. Humans make purchasing decisions based on their previous experiences. They tend to learn from both successful and unsuccessful experiences. In case of using 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. Similarly, agents in this framework, as in 43, use previous experience and adjust their attitude in order to gain successful negotiation.

Another important aspect borrowed from 43 into this model 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 an e-commerce system cannot have these facilities, they should rely on some external resources. Ontology used in this model provides agents with background information in the form of public price range so that these agents can make decisions bounded by those price ranges.

Another important aspect of this framework includes the dynamic decision making model that enables agents to readjust their negotiation strategies when there is a change inside or outside the e-commerce environment. Humans' negotiation tactics tend to adjust dynamically with respect to the changes that occur in the real world. Just like humans change their negotiating price range when there is a change in supply demand ratio, buyer and seller agents in an e-commerce marketplace should also dynamically change their negotiating price range when there is an increase or decrease in number of agents in the marketplace. Incorporation of BDI model of agency into this model accomplishes such task in this framework.

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 experiences and making dynamic decisions during negotiation process.

5.2 Experiments and Results Provided in 43

5.2.1 Experiments

The experiments and results provided in 43 focuses on how the agents behave in familiar situations (similar situations encountered earlier) and in multiple interactions using their

experience and attitude. There are two categories of scenarios considered in testing the behavior of the agents. One category of scenarios is various experiences and the other is various degrees of interaction. In testing the use of experience, following scenarios ware considered to evaluate the model:

- Using a successful experience of buying or selling the same product
- Using an unsuccessful experience of buying or selling the same product
- Using a successful experience of buying or selling a similar product, not the same product in question
- Using an unsuccessful experience of buying or selling a similar product, not the same product in question.

Negotiating participants' decisions are affected by number of participants involved during the negotiation. Therefore, degree of interaction is an important factor that affects agent behavior with respect to their attitude. In testing the use of degree of interaction, following scenarios ware considered:

- One buyer and one seller
- One buyer and many sellers
- Many buyers and one seller
- Many buyers and many sellers

According to 43's believes, the above two categories subsume the most plausible situation in human buying behavior.

5.2.2 Results

In 43, various scenarios of experiences and various degrees of interactions are experimented. The claim that an agent with buying or selling relevant experience should be able to use that experience when buying or selling a similar product is proved with their presented experiment results. Results in 43 show that the above claim is true. Moreover, their results show that agents pick up the best possible experience from successful and unsuccessful experiences in solving a similar problem. Furthermore,

changes in Case Base system suggest that agents update or refine their experiences accordingly.

Another observation given in 43 is that many-to-many interactions may help agents to finish their negotiation faster. When there is one-to-one negotiation these agents don't have many choices, so they continue to negotiate until an agreement is reached or they abandon the negotiation if their interest are not met. Whereas in many-to-many interactions agents have more choices and they tend to be more selective in making deals, for their benefit.

5.3 Need for Dynamic Negotiation

In today's e-commerce environment, buyer and seller agents can dynamically enter and leave the marketplace with external knowledge of a certain product. The agents that are already negotiating for the same product in the system should somehow become aware of any external change occurred for that product. Becoming aware of the changes that happened inside or outside the system would affect the negotiation process of the agents who are already negotiating for that product. The ideas, experiments and results considered in 43 fail to consider the above dynamic environment situation. In fact, 43 acts like a task oriented conventional system by negotiating for the product with inaccurate information without giving any consideration to the changes that occurred in the environment. Therefore, there is a need to extend 43 to handle dynamic changes that happens inside or outside the system so that the agents in the system can be competitive in today's uncertain e-commerce marketplace.

5.4 Scenarios considered for experiments

Evaluation of this framework is complex as there are many parameters, including dynamic environment changes to be considered. 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 real world buying and selling situations. The focus of this framework is on how the agents behave to internal and

external dynamic changes with their attitude and experience. Testing these issues in experiments needs careful selection of buying and selling scenarios and cases, so that they satisfy constrains in the proposed model and also reflect real world situations. We consider two scenarios in our experiments towards the aim of testing the behavior of these agents in making successful or unsuccessful deals.

5.4.1 Scenarios with internal changes (SDR)

Users create agents at their own will and send them to the marketplace for negotiation. Agents on the other hand leave the marketplace when they finish their negotiation. This involuntary process of both users and agents create high uncertainty in supply demand ratio for a product in the marketplace. Dynamic change in supply demand ratio is an important factor in the negotiation phase of an agent. When testing such dynamic factor that takes place inside the e-commerce system, one should consider the following scenarios:

- Effect of increase in SDR for a buyer agent
- Effect of decrease in SDR for a buyer agent
- Effect of increase in SDR for a seller agent and
- Effect of decrease in SDR for a seller agent

We believe that above scenarios address the entire possible dynamic supply demand ratio changes in an e-commerce marketplace. Experiments involving these combinations are good measure of evaluating the behavior of dynamically negotiating agents in this framework.

5.4.2 Scenarios with external changes (PPR)

Negotiation process of agents in an e-commerce system can be affected by many external factors. A company's reputation or the quality of a product for example may play a major role in changing the public price range of a product. Such uncertain change that can happen outside the e-commerce system may create high degree of uncertainty in the price range within which agents negotiate in an e-commerce system. Therefore, such external

changes that can affect the negotiation process of agents in an e-commerce system should be considered as an important factor in the negotiation phase of an agent. When testing such dynamic factor that takes place outside the e-commerce system, one should consider the following scenarios:

- Effect of increase in PPR for a buyer agent
- Effect of decrease in PPR for a buyer agent
- Effect of increase in PPR for a seller agent and
- Effect of decrease in PPR for a seller agent

We believe that above scenarios illustrate how to compensate for one of many external factors that can affect a negotiation process of agents in an e-commerce system. Experiments involving these combinations are good measure of evaluating the behavior of dynamically negotiating agents in this framework.

5.5 Experiments and Results

The experiments considered here relate to two important issues on which the dynamic negotiation process of human buying behavior is dependent in general: 1) scenarios with internal changes 2) scenarios with external changes as discussed in last section.

5.5.1 Experiments testing internal changes

Our assumption about internal changes that affects the negotiation process can be stated as follows:

"Agents that are negotiating with attitude and relevant experience should be able to adjust their negotiating strategies to compensate for internal changes"

To evaluate this assumption we need to allow agents to enter and leave the marketplace when there are other agents negotiating for the same product. The agents who are already in the marketplace, recalculate their negotiating price range when a new agent enters or leaves the marketplace. The result of the experiment presented in Appendix F proves the above claim. Note that we allowed negotiating agents to recalculate their price ranges only when supply demand ratio changes by at least 10%. In the real world, minor changes

in supply demand ratio doesn't affect the negotiation strategies of human negotiators. This is the why we did not allow negotiating agents to recalculate their price ranges when the change in supply demand ratio is less than 10%.

The assumption that *"Agents that are negotiating with attitude and relevant experience should be able to adjust their negotiating strategies to compensate for internal changes"* is valid as the above discussion supports it. The result from the experiment also supports that these agents do change their negotiating price range to compensate and be competitive to any supply demand changes that occurs in the marketplace.

5.5.2 Experiments testing external changes

Our assumption about external changes that affects the negotiation process can be stated as follows:

"Agents that are negotiating with attitude and relevant experience should be able to adjust their negotiating strategies to compensate for external changes"

To evaluate this assumption we need notify the external changes, such as public price range change, to the negotiating agents of the e-commerce system. The negotiating agents in the marketplace recalculate their negotiating price range when they are notified of change in public price range. The result of the experiment presented in Appendix F proves the above claim.

The assumption that *"Agents that are negotiating with attitude and relevant experience should be able to adjust their negotiating strategies to compensate for external changes"* is valid as the above discussion supports it. The result from the experiment also supports that these agents do change their negotiating price range to compensate and be competitive when there is a change in public price range outside the system. The observation of this experiments infer that the agents in this framework use attitude, relevant experience and dynamic negotiation strategies to perform dynamic negotiation.

5.6 Computational issues

Complexity of the proposed framework can be verified at several stages of the execution phase. The stages that we consider in calculating the complexity of the framework are i) retrieval of domain specific information ii) retrieval of experience from Case Base iii) creation, interaction of clones and negotiation process and iv) recalculation during the dynamic changes in the environment.

Retrieval of domain specific information from Ontology takes a cost of an SQL query as the Ontology is implemented directly in a database. However, in case of a formal Ontology, the complexity of retrieving domain specific information will be the number of accesses to the information in the Ontology hierarchy tree. Retrieval of relevant experience from Case Base in this system is linearly proportional to the number of cases existing in the Case Base. However, if there is vast number of cases in a Case Base, indexing can be used to reduce this complexity considerably. Complexity of the entire process of negotiation is m X n, where m is number of unique buyers and n is number of unique sellers for the same product in the marketplace. This is because each buyer creates as many clones as unique sellers and each seller creates as many clones as unique buyers in the marketplace. Since finding offer and counteroffer in our implementation do not take heavy computation, they are considered as done in constant time. Recalculation of price ranges as well as broadcasting new price ranges to the clones are also considered to be done in constant time as they do not take considerable computation in the implementation of the framework.

Chapter 6

Conclusion and Future Work

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

The agents in this framework, as in 37, 38 and 43, have attitude that represents few aspects of mental state of its clients. They learn from experiences using Case Based Reasoning techniques and reason accordingly during the negotiation process. Moreover, these agents are capable of dynamically handling uncertain changes that happens inside and outside the application environment while they participate in negotiation. The interaction among these competing and cooperating agents can be one-to-one, one-to-many or many-to-many. Java programming language is used as the language of implementation. The most important aspect of this framework lies in its compensating capability to uncertain changes in e-commerce environment during negotiation by using the BDI model of agency principle.

The experiments presented in Appendix F demonstrate that it is possible to build a framework with limited aspects of dynamic negotiation among competitive and cooperative software agents for an e-commerce system. Further, it shows that it is possible to incorporate few important aspects of mental state of the clients and also to make use of previous experiences in this dynamic negotiation model. The adjustments took place in Case Base during the experiments suggests that agents in this framework learn and also refine their experiences as they involve in dynamic negotiation.

7.1 Future Work

Currently, implementation of this proposed framework is done in a single sequential machine. This should be extended to a distributed environment, where agents from different marketplaces could negotiate in parallel, in order for this e-commerce framework to become a reality. Moreover, the weights assigned in the fuzzy functions of this and the previous model [37] needs a through study before it can be used in the real world. There is a need to focus on the offer and counter-offer techniques of this model. Perhaps the ideas presented in [15] (Black Board approach) or the ideas presented in [42] (using internal Beliefs) may improve the negotiation tactics of this system.

Bibliography

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

2. Anand S. Rao, Michael P. and Georgeff. 1995. **BDI Agents: From Theory to Practice.** *Proceedings of the First International Conference on MultiagentSystems, Technical Note 56, ICMAS 1995.*

3. Anand R. Tripathi, Neeran M. Karnik, Ram D. Singh, Tanvir Ahmed, John Eberhard, Arvind Prakash. 1999. **Development of Mobile Agent Applications with Ajanta.** *Technical Report. Department of Computer Science, University of Minnesota, Minneapolis.*

4. Alessio Lomuscio, Michael Wooldridge, Nicholas Jennings. 2000. **A classification scheme for negotiation in electronic commerce.** *Journal of Group Decision and Negotiation 19 – 33.*

5. 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).*

6. Charles J. Petrie. 1996. **Agent Based Engineering, the Web, and Intelligence.** *IEEE Expert (December 1996).*

7. Damir Horvat, Dragana Cvetkovic, Veljko Milutinovic, Petar Kocovic, Vlada Kovacevic. 2000. **Mobile Agents and Java Mobile Agents Toolkits.** *Proceedings of the 33rd Hawaii International Conference on System Sciences (Maui, Hawaii, USA).*

8. David Kotz, Robert S. Gray. 1999. **Mobile Agents and the Future of the Internet.** *ACM Operating Systems Review, 33(3), Pages 7-13.*

9. David Wong, Noemi Paciorek, Dana Moore. 1999. **Java-Based Mobile Agents.** *Communications of the ACM vol.42, No.3.*

10. 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).*

11. H. Vogler, A. Buchmann. 1998. **Using Multiple Mobile Agents for Distributed Transactions.** *3rd IFCIS Conference on Cooperative Information Systems (CoopIS'98) (New York City, USA, August 1998).*

12. Hyacinth S. Nwana, Jeff Rosenschein, Tuomas Sandholm, Carles Sierra, Pattie Maes, Rob Guttmann. 1998. **Agent-Mediated Electronic Commerce: Issues, Challenges and some Viewpoints.** ***Autonomous Agents '98, Proceedings of the Workshop on Agent Mediated Electronic Trading (AMET '98).***

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

14. J.D. Tygar. 1998. **Atomicity in Electronic Commerce.** *Mixed Media, April/May 1998, pages 32-43.*

15. 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).***

16. Jonathan Bredin, David Kotz, Daniela Rus. 1998. **Market-based Resource Control for Mobile Agents.** *Autonomous Agents '98, Minneapolis, MN, USA.*

17. Lai, R. and Menq-Wen Lin. 2002. **Agent negotiation as fuzzy constraint processing.** In *Proceedings of the 2002 IEEE International Conference on Fuzzy Systems, 2002, pages 1021 - 1026*

18. 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.*

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. Mehdi Jazayeri and Wolfgang Lugmayr. 1998. **Gypsy: A Component-Based Mobile Agent System.** *Proceedings of the 8th Euromicro Workshop on Parallel and Distributed Processing.*

21. Michael Georgeff, Barney Pell, Martha Pollack, Michael Wooldridge. 1998. **The Belief-Desire-Intention Model of Agency.** *Proceedings of the 5th International Workshop on Intelligent Agents 5: Agent Theories, Architectures, and Languages.*

22. 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.*

23. 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.

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

25. 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.*

26. 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.*

27. Prithviraj Dasgupta, Nitya Narasimhan, Louise E. Moser, P.M. Melliar-Smith. 1999. **MAgNET: Mobile Agents for Networked Electronic Trading.** *IEEE Transactions on Knowledge and Data Engineering, Vol.11, No.4. july/august 1999.*

28. R. Guttman, A. Moukas, and P. Maes. 1998. **Agent-mediated Electronic Commerce: A Survey.** *Knowledge Engineering Review, Vol. 13:3, June 1998.*

29. 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.*

30. 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.*

31. Robert J. Glushko, Jay M. Tenenbaum and Bart Meltzer. 1999. **An XML framework for agent-based E-commerce.** *Communications of the ACM, Volume 42, No. 3 (Mar. 1999).*

32. Sebastian Abeck, Andreas Köppel, Jochen Seitz. 1998. **A Management Architecture for Multi-Agent Systems.** *Proceedings of the IEEE Third International Workshop on Systems Management.*

33. 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.*

34. 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.*

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

36. Walid S. Saba and Pratap Sathi. 2001. **Agent Negotiation in a Virtual Marketplace.** *In Proceedings of the 2nd Asia-Pacific Conference on Intelligent Agent Technology, 2001, pages 444-453. IAT.*

37. Walid Saba. 2001. **Modeling Mental States in Agent Negotiation.** *Negotiation Methods for Autonomous Cooperative Systems 2001 Fall Symposium, Pages 142 – 147, AAAI Press.*

38. Walid S. Saba. 2002. **Negotiating with an attitude in a virtual marketplace.** *International Journal of Computational Intelligence and Organizations* (to appear)

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. Wong, W. Y., Zhang, D. M., and Kara-Ali, M. 2000. **Negotiating with experience.** *Knowledge-Based Electronic Markets, Technical Report WS-00-04, PP. 85-90. AAAI 2000.*

41. Mudgal C. and Vassileva J. **Multi-Agent Negotiation to Support an Economy for Online Help and Tutoring.** *Proceedings the 5th International Conference on Intelligent Tutoring Systems, Pages 83-92, Springer LNCS 1839.*

42. Pu Huang and Katia Sycara. 2002. **A Computational Model For Online Agent Negotiation.** *Proceedings of the 35th Annual Hawaii International Conference, Pages 363 – 369, IEEE '02.*

43. Pratap R. Sathi. 2001. **Experienced agents with attitude in a virtual marketplace.** *Thesis, University of Windsor, 2001.*

Appendix A

Electronic Commerce

Electronic Commerce (e-commerce) is defined as the conduct of commerce of goods and services with the assistance of existing technologies over the Internet [12].

A.1 Concept of E-Commerce

E-Commerce concepts include Business-to-Business e-commerce (B2B), Business-to-Consumer e-commerce (B2C) and Consumer-to-Consumer e-commerce (C2C). B2B is the use of private networks on the Internet to automate business transactions between companies. B2C e-commerce is a retail sale model or a web market. Amazon.com is an example of B2C. It enhances business models by offering:

- A global audience
- Unlimited product selection
- Portal sites that refer consumers to the actual purchasing site
- Focused marketing that can be quickly tailored to consumer

C2C e-commerce is an auction based or bargain-based systems [28]. It often provides low cost consumer-to-consumer refurbished goods to reduce transaction cost. Some popular examples of C2C e-commerce sites are eBay.com, uBid.com and eWanted.com.

A.2 Properties and Requirements of E-commerce transactions

Different conferences on Principles of Distributed Computing (PODC) have agreed on certain concepts that are heavily used in electronic commerce [14, 18].

- Atomic transactions.
- Providing support for a variety of transaction type including simple buying and selling, auctions and complex multi-agent contract negotiation.
- Cryptographically secure protocols.

- Providing language in which the rich array of semantic content about commerce is expressed.
- Being extensible, by third parties, so providing multi-agent contract and dynamic mediation.
- Providing a secure and private credit and payment mechanisms.
- Interoperating with other new and existing E-commerce service and
- High reliability.

Apart from the issues mentioned above, electronic commerce encompasses a broad range of issues like reputation, law, advertising, ontology, intermediaries, multimedia shopping experiences and back office management [28].

A.3 Consumer Buying Behavior (CBB)

In commerce and in tern e-commerce, different models of CBB share a similar list of six fundamental stages in guiding consumers [28]. The six stages can be summarized in the following:

- *Need Identification*: consumer becoming aware of some unmet need.
- *Product Brokering*: Retrieval of information to help determine what to buy.
- *Merchant Brokering*: Merchant specific information to help determine who to buy from.
- *Negotiation*: How to determine the terms of transaction including price bargains, warranties etc.
- *Purchase and delivery*: It can be a signal to the termination of negotiation stage or occur sometime after the negotiation is done.
- *Service and Evaluation*: this phase involves product service, customer service etc.

A.4 Challenges in e-commerce

There are many systems in existence with different models for Product Brokering, Merchant Brokering and Purchase stages. Recommender systems as discussed in [34] use collaborative filtering and knowledge-based approach to make recommendations to the

users in purchasing decisions. There are other Recommender systems, like PersonaLogic and Firefly that help consumers find the products. Firefly recommends products using automated collaborative filtering approach. Systems like BargainFinger, Jango, and Kasbah helped in the Merchant brokering stage. But there are not many systems, which could support the negotiation stage. MIT's Kasbah [5] was one of the first systems to support the automated negotiation stage. Though it had its own drawbacks, Kasbah led the other researchers to work in this negotiation aspect more actively.

There are several challenges or reasons for not many automated negotiation systems in e-commerce. The challenges of automated negotiation in E-Commerce applications as discussed in [15] are:

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 for item.
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.

Reaching the challenges mentioned above brings in the issues like interoperation and automation. For example, to automate negotiation in buying and selling a car, there needs to be a semantically interoperable language and protocol coordinating the parties (agents) involved. Unfortunately, there is still lack of common language and ontology for e-commerce interoperation. Although HTML web-scraping may get us by for certain problems, for instance, product information retrieval in retail markets, it is not sufficiently robust to base important business processes upon [12]. Extensible Markup Language (XML) came as a good tool in differentiating products from more than just their prices. It helped merchants to describe various services offered with the product sold, eliminated the need for web scraping by the use of XML parser and brought in the XML/EDI message format reducing the cost of transactions in e-commerce [31]. Nonetheless, there are still problems, which will take much more effort on business corporations in agreeing on Meta Tags in XML to specific semantics in accomplishing

the tasks mentioned above. Business ventures are coming up with Business Interface Definitions (BDI) and Common Business Library (CBL) as domain specific ontologies to accomplish this cumbersome task [31].

Appendix B

Agents

Agent based computing is a recent approach to problem solving in complex heterogeneous systems that has been attracting great deal of attention among the Artificial Intelligence (AI) community. People have been fascinated about the idea of artificial agencies for a long time. Especially, with the inception of AI and distributed systems, computer scientists have been working on systems which could automatically perform tasks for humans.

B.1 History of Software Agents

Since the beginning of Artificial Intelligence, Object technology and Distributed

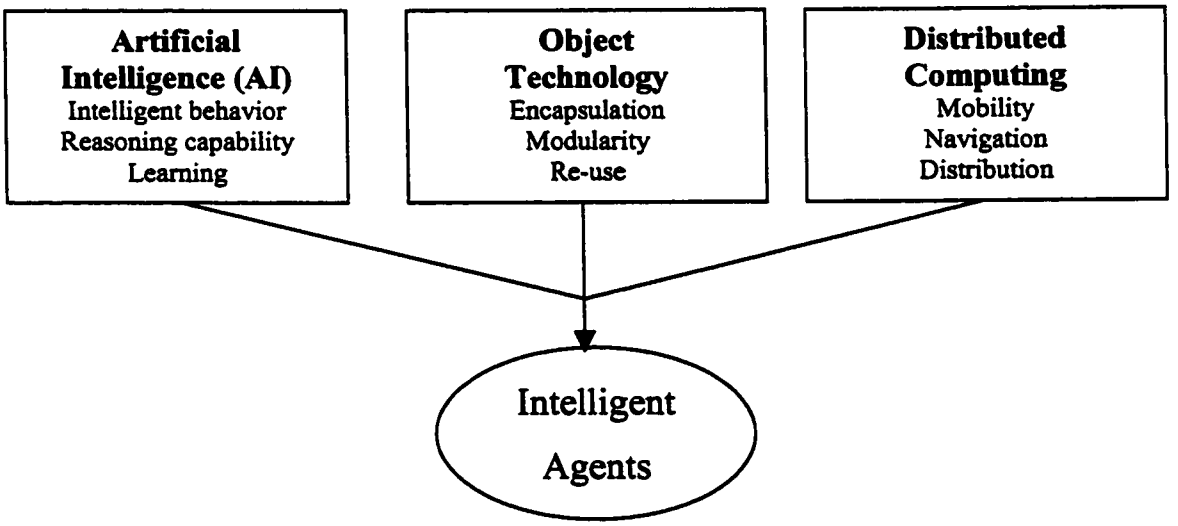

Figure B.1: Evolution of Intelligent Agents

Systems, as illustrated in Figure B.1, momentum has shifted from hardware to software. Researchers have been working extensively to build intelligent software agents to perform tasks that are only performed by humans.

Even after two decades of research, some of the key concepts in agent-based computing lack universally accepted definitions. Embarrassment comes as no surprise to the Agent

community, as they still cannot agree on "what is an agent". There are two main reasons why it is difficult to precisely define what an agent is. First, agent is a term that is widely used in everyday parlance as in travel agents, estate agents, etc. Second, even in the software fraternity, the word agent is really an umbrella term for a heterogeneous body of research and development. The confusion about agents led researchers to invent more synonyms including knowbots (knowledge-based bots), softbots (software robot), taskbots (task-based robots), userbots, robots, personal agents, autonomous agents (mobile agents), auctionbots and personal assistants [13].

B.2 Definition of an Agent

One of the most acceptable definitions for agents by two prominent researchers Jennings and Woodridge in Software Agent Technology states [22]:

> *"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."*

As per this definition, Agents are: (i) clearly identifiable problem solving entities with well-defined boundaries and interfaces; (ii) situated in a particular environment and they observe the state of the environment; (iii) designed to fulfill a specific role; (iv) autonomous, have control on both their internal state and over their own behavior; and (v) capable of exhibiting flexible problem solving behaviors [22].

B.3 Types of agents

Agents may be classified based on the mobility factor, i.e., *static* or *mobile*. Another way of classification is *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 behavior by responding to the present state of environment in which they are embedded [13].

Agents interact with its environment, sometimes even out of its environment (in case of mobile agents) for enhancing collaboration among them. Combing several of such agents pursuing the same goal leads to the multi-agent systems (MAS) [32]. Multi-agent systems allow for scalability, permit software reuse and handle software evolution.

There is a close relationship between multi-agent systems and mobile agents. Mobile agents are programs that can migrate from host to host in a network, at times to places of their own choice. The state of the running program is saved, transported to the new host, and restored, allowing the program to continue where it was left off [8]. Mobile agents are an effective choice for many applications, since they improve latency and bandwidth of client-server applications and reducing vulnerability to network disconnection. They support transactions in massively distributed environments, support systems which involve electronic cash and banking systems [16], support activities in dynamically changing environments, support mobile devices and coordination of different types of applications and resources [11].

B.4 Implementation of Agents

Java is known to be an effective implementation language for mobile agents. Most of the present Mobile agent frameworks like Ajanta [6], Concordia, Odyssey, JAFMAS [3], Voyager, MAgNET [27], Gypsy [20] and IBM's Aglets [7] are all implemented in Java. Multi platform support and the promise of write-once and run-anywhere operation make Java extremely suited for mobile agent technology [9]. Java's object serializations accomplish the conversion of an agent and its state into a form suitable for network transmission and allow the remote system to reconstruct the agent. Some Java- based mobile agents systems also provide persistent agent state information. Java also facilitates the migration of code and state via its class-loading mechanism. Java based mobile agent systems are the best choice for e-commerce applications [9].

Appendix C

Automated Negotiation

Automated negotiation is becoming an integral and important part of e-commerce. Real world negotiations in general accrue transaction costs and time that may be too much for both merchants and consumers alike. A good automated negotiation can both save time and find better deals in the complex and uncertain business environment [39]. However, current e-commerce environment only supports non-interactive buying-selling types of auction and there are no models yet for automated negotiation in E-Commerce.

Research area that merges negotiation with software-agents is the broad field of Multi Agent System (MAS). In MAS, there is no global control, no globally consistent knowledge, and no globally shared goals. They are concerned with coordinating intelligent behavior within a collection of autonomous (possibly heterogeneous) intelligent agents. MAS assume total self-interest and a high degree of *competition* among agents during negotiations for limited resources [28]. This behavior of MAS best suites our needs in e-commerce environments.

Agents have a high degree of self-determination, since they decide for themselves what, when and under what conditions their actions should be performed. In an e-commerce environment such agents need to interact with other autonomous agents to achieve their objectives. Since agents do not have direct control over one another, they must persuade other agents to act in particular ways to achieve their goals. This concept of persuasion is called *negotiation*; a process by which agents come to a mutually acceptable decision on some matter [24].

When faced with the need to reach agreements on a variety of issues, humans make use of negotiation process. Similarly, automated negotiation can become a fundamental operation for shopping agents in e-commerce. This automation of negotiation can significantly reduce the time it takes to negotiate, making large volume of transactions

possible in a small amount of time. This can also remove some of the discretion of humans to engage in negotiation, for example, embarrassments and personality manipulations. For these reasons, the formalisation of negotiation has received a great deal of attention from the multi agent systems community throughout the past two decades.

C.1 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. Game Theory in economics describes such an interaction in terms of protocols and strategies [108]. There are two important theorems exists on negotiation. One is Game theory and the other is Epistemic Logics.

Game Theory

Game theory views an agent as an individual, a firm or some complex organization where the functionality of the agent is profit maximizing. Game theory models do not describe how the world is or must be, but they describe how the world could be. An out come of a game is usually decided by the information in the structure and the strategy used in the game. Various criteria of individual optimality in game theory include Dominance, Nash Equilibrium, Bayesian Nash Equilibrium, trembling hand equilibrium, and sequential equilibrium. Nash equilibrium is the 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 best response to the predicted strategies of other agents and it should maximize the utility or profit [35].

Epistemic Logics

Distribution and transfer of information among autonomous agents are essential characteristics of many environments. Representing the information and reasoning the state of the information while taking into account of the dynamics of the information is the core idea behind analyzing environments. The formalism that support such representation and reasoning are called 'epistemic logic' or 'logic of knowledge.

Dynamics of distributed systems can 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 represents the knowledge about the system.

Another definition for automated negotiation by Jennings et al. [23] 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 and attempts to cooperate and coordinate and is required both when both agents are self-interested and are cooperative.

Although various disciplines have proposed different theorems on negotiation, it is clear that negotiation theory covers a wide range of phenomena encompassing different approaches such as Artificial Intelligence, Social Psychology, and Game theory. Negotiation research can be considered to deal with three broad topics [23].

1. **Negotiation Protocols:** These are the set of rules that govern the interaction. These rules cover permissible types of participants, the negotiation states, the events, what can 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 hundreds of 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 protocols in place, the nature of the negotiation object and the range of operations which can be performed on it.

Given a wide variety of possibilities, including game theory and epistemic logics given above, there are no universally accepted approach or technique for agent negotiation. The minimum capabilities required for an automated 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. However, if agents can only accept and reject others proposals, then negotiation can be very time consuming and inefficient. This results in the proposer having no means of ascertaining whether 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.

Negotiating strategies to reach an agreement often depend on the specific issues or parameters under consideration. For instance, whether merchandise has a common value or whether it differs from agent to agent may call for different negotiation strategies to reach an agreement. Negotiation mechanism consists of a negotiation protocol coupled with the negotiation strategies for the agents involved. There are some properties that are generally considered desirable for negotiation mechanism [4]:

- **Computational efficiency:** Concerned with the need a negotiation mechanism that is computationally efficient. In other words, computational costs carried out at run-time must be manageable.
- **Communication efficiency:** Concerned with having a mechanism that handles communication among the agents in an efficient way.
- **Individual rationality:** Mechanism that satisfies individual rationality for all the agents involved in negotiation. In other words, agent's independent interest to participate in negotiation.
- **Distribution of computation:** Mechanisms that distribute the computation over the agents are preferable to the ones in which one server is performing all the computation for the whole system. This is due to the desire to avoid the disruptive effects of a single point of failure and performance bottlenecks.

C.2 Parameters of Negotiation

We have seen in the last section that negotiation deals with negotiation protocols, negotiation objects and negotiation decision models. Negotiation objects or number of issues involved in a negotiation can play a crucial role in determining negotiation

strategy. There has been a tremendous amount of effort put in identifying the parameters on which *any* type of negotiation can take place [4].

C.2.1 Cardinality of the Negotiation

There are two important issues in cardinality of negotiation parameter, namely negotiation domain and the interaction type.

- Negotiation domain: single-issue or multiple-issue; and
- Interactions: one-to-one, many-to-one, many-to-many.

Domain of negotiation can be visualized as set of tuples over which the agents negotiate to reach agreement. Each elements of this tuples may represent an issue such as price, quality, warranties, delivery time, and so on. When we have only one issue in a negotiation, for example price, the tuples are singletons. In multiple issue negotiation, different issues might be related by some publicly agreed utility function [4].

Interactions between agents can be classified based on a number of agents involved in the negotiation. One-to-one negotiation in which one agent negotiates with exactly one other agent becomes important due to the business-to-business e-commerce scenarios. Many-to-one negotiation where many agents negotiate with one agent is exactly same as auction setting. In this case, one agent plays the role of the seller and the rest play the buyer's role. Many-to-many negotiation where many agents negotiate with many other agents creates the most complex scenario of all interactions [4].

C.2.2 Agent Characteristics

In a sense we can agree that agents are nothing but computational entities that participate in negotiation processes that must be capable of rating its preferences to evaluate and choose between number of deals. Further characterizations of agents are:

- **Role:** Agent's types are the role that they play in the negotiation (buyers, sellers, or both). Usually buyers and sellers are the important roles but in case of auction negotiation, intermediaries can have an important role as well.

- **Rationality:** Rationality can be perfect or bounded. Game theory (discussed later in this Chapter) assumes perfect rationality meaning that large computations can be performed at a constant time. However, in practice, agents are forced to bid or withdraw because they do not have the computational power. Thus, negotiation models that assume perfect rationality have to use approximations in practice, whereas models that explicitly assume bounded rationality are more realistic in this sense [4].
- **Knowledge:** Private information such as internal deadline and utility functions are important parameters for agents. Whether an agent holds private information or not will directly affect the agent's bidding strategy.
- **Commitment:** When an offer is made, agents may wait until an acceptance or counter-offer is received. Alternatively, the agent can bid to other agents without waiting. Thus, there can be various levels of commitment placed in the protocol [4].

C.2.3 Environment and Goods Characteristics

The negotiation environment can be either static or dynamic. Dynamicity of the environment can affect the utility function of the agents in a delicate way. Utility function reflects the preference of an agent. While in a static environment an agent does not learn during the process and maintains a fixed utility function, this behavior would be less likely to produce a positive payoff in a very dynamic environment.

The characteristics such as private or public value of the goods also crucially define the negotiation protocol. The values of goods depend on whether it will be used for private (e.g., a cake) or public (e.g., bonds). For example, when buying a car, both the buyer's preferences and how the car will preserve its value over time should be considered if one is interested in selling the car.

C.2.4 Event Parameters

The negotiation protocol is mainly influenced by the ways in which the events take place during the negotiation.

- **Clearing schedule and timeouts:** is an event producing a temporary allocation between buyer and seller. Clears can be scheduled at random or following other events. For example, during the bidding phase of an auction each round terminates with a temporary allocation of the good being auctioned to the prospective buyer that meets the auctioneer's call. Timeouts determine the closing of the negotiation; therefore, they transform clears into "final clears", i.e., a final agreement between buyer and seller about the transaction.
- **Quotes Schedule:** Often third-party quotes are generated through the Recommender systems and they need to be regulated. Otherwise, too many requests for quotes can significantly slow down the negotiation mechanism.

C.2.5 Information Parameters

Information or messages other than bids can be passed between agents before and during negotiation to help buyers and sellers reach an agreement. Such messages can be beneficial in order to save computational time of the agents. Among many useful messages, we will look at two important such messages.

- **Price quotes:** Quotes generated by potential buyers requesting an analytical price from a seller before starting a negotiation can be useful to all parties as they reduce negotiation time.
- **Transaction history:** History of transactions given or requested by buyers and sellers can increase the credibility involved in negotiation. Together with the trusted third party quotes, transaction histories can form the basis for argumentation-based negotiation. Expert human negotiators often focus on the reasons why an offer is not acceptable and try to persuade their counterparts to the characteristics an agreement will have to include in reaching the deal. Artificial negotiators propose offers to counterpart but they do not try and motivate them to reach an agreement. This is often seen as a severe limitation that can limit the flexibility of the negotiation.

C.3 Negotiation Process

Additional feedback on a proposal that indicates more than whether an agent agrees with the proposal or not can improve the efficiency of the negotiation process. This feedback can take the form of a critique or a counter proposal. From such a feedback, the proposer should be able to bring the recipient more close to the agreement space. Achievement of reaching the agreement quickly in tern depends on the intelligence (reasoning) of the agents involved. These agents will have to follow different strategies and negotiation algorithms and a family of negotiation tactics. When we equip these agents with these negotiation techniques and tactics, they can negotiate at any place such as classified negotiation, stock market negotiation and retail auction negotiation [29]. There have been impressive results from MIT's Kasbah Agent market place [30] and Agents with attitude in a Virtual Marketplace [36], where some of the negotiation strategies have been modeled and implemented.

There have been different approaches proposed for integrating intelligence factors like negotiation strategies, tradeoff mechanisms [25] [26], different negotiating functions [1] and tactics in these agents. *Reinforcement learning* is one approach based on rewarding actions that turn out to be positive and punishing those that are negative [10]. *Rule based learning* is another approach for negotiating agents in a virtual market place [18] which is based on particular rules in the system that is proved to be effective. Case-Based Reasoning (CBR) is yet another approach for negotiation where we capture and reuse successful negotiating experiences.

C.4 Challenges In E-Commerce Negotiation

As we have seen before, 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 but the last one seems to be more complex and it depended on cardinality of negotiation, agent characteristics, environments and goods characteristics, event parameters and information parameters. These parameters may vary form domain to domain. Therefore, in most cases negotiation strategies and tactics are completely domain

dependant. In e-commerce set up, negotiation gets even more complex, as the parameters here are fuzzy, dynamic and vary diverse. The challenges for automated negotiation in e-commerce applications include the following [15]:

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-agent 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. Simply put, there is a need for development of better domain suited negotiation techniques.

Appendix D

Case Based Reasoning

Case-Based Reasoning (CBR) helps solve new problems by remembering a previous similar situation and by reusing those 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 based on *Theory of Dynamic memory* proposed by R. Schank. This theorem states that understanding takes place by integrating of new things encountered with old experiences or what is already known (experience). Also it states that Understanding causes us to remember old experiences, consciously or unconsciously, as we process the new ones. Moreover the theorem states that remembering, understanding, experiencing and learning cannot be separated form each other.

There are many successful case-based reasoning systems 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.

From the theory of Dynamic memory we can infer that 'case-based reasoning' is both cognitively plausible model of reasoning and a method for building intelligent systems. This system integrates problem solving, understanding, learning and memory in to one framework.

Appendix E

Belief, Desire and Intention model of Agency

Most of the conventional software applications are designed for static world. These conventional software are assumed to work with perfect knowledge, meaning that they have all the information they need to make their decisions. However, in the real world, these systems are embedded in dynamic environments. Therefore, when it comes to real world issues, they have only *partial information* available for them to make any decision i.e., their access to dynamic information is limited. Moreover, the systems in existence don't have unlimited computational recourses [21].

Belief, Desire and Intension (BDI) decision-making model is interested in solving dynamic and uncertain environment problems. BDI model has become the best-known and best-studied model of practical reasoning agents. There are several successful applications exist based on BDI model. Fault diagnosis system for space shuttle and factory process control system are couple examples where BDI model of agency is used [2].

Belief represents the knowledge of the world. Computationally, Believes are some way of representing the state of the world. For example, belief in a BDI system could be a value of a variable or topples of a rational database. Hence, Belief represents information about the world. Belief is needed because the world changes and we need to remember the past events. The reason why we need to remember the past events is because if I want to get somewhere for example, I need to know where I am right now in order to find out how I can get there.

Desire, or more commonly, the Goal. Computationally, it may be a value of a variable or a symbolic expression in some logic. The important point is that a Goal should represent some desired end state. Conventional systems also has desired end state but they are 'task oriented' than goal oriented. This means the system cannot automatically recover from

failures. For example, the reason we recover from a missed train is because we know where we are (through our believes) and we remember to where we want to go (through our Goals). Task oriented conventional software would fail in this above situation but BDI model would not.

Belief and desire enables to decide on a plan to achieve the goal but in a dynamic environment, where there can be a change in the environment that could affect the achievement of the goal, what should we do? Classical decision theory states that we should always replan when there is a change in the environment. In contrast, the conventional system goes on executing the tasks with no consideration to the changes in the environment. There are problems associated with both the ideas. A system cannot ignore the changes and execute the tasks in a dynamic environment nor it cannot replan for every single change in the environment because of the limited resources.

The third component of the BDI model, **Intention** states that 'the system needs to commit to the plans and sub goals it adopts but it must also be capable of reconsidering these adopted plans at appropriate (crucial) moments. Computationally, Intensions may be simply be a set of executing threads in a process that can be appropriately interrupted upon receiving feedback from the possibly changing world. For example a *Flight Scheduler agent* which schedules Arrival Time of the flights on the runway in an airport may have number of threads running for each flights. As the wind or weather changes (these are the environment changes) the process should be appropriately interrupted so that the process can replan and inform the respective flight and make sure the flight can still reach the runway at the expected time [2].

A negotiation in a real world, either cooperatively or competitively, takes place in an uncertain and dynamic environment. This BDI model of agency theory gives us a realistic decision solution to such dynamic environment negotiation problems.

Appendix F

Results

Testing the framework starts with creating data for Ontology and Case Base. First, we populate the Ontology database with information on products and conceptual similarity. Then we populate buyer and seller experience databases with number of cases. Following two tables are used in testing this framework:

PCat	PN	Warr	PPR	Price	Attitude	SDR	NegRec	Res
comprods	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	1
homeelec	TV	1	300-500	420	.7,.5,.7	1.5	310/345/380/420	1
comprod	printer	1	300-450	335	.2,.8,.3	.5	305/320/335	0

Table F.1 Buyer Experiences in database

PCat	PN	Warr	PPR	Price	Attitude	SDR	NegRec	Res
comprods	printer	1.5	300-450	360	.8,.3,.9	2	420/390/360	1
comprods	mouse	1	8-15	10	.4,.1,.8	3	15/13.5/10	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
comprod	printer	1	300-450	420	.1,1.0,.3	2	450/440/430/420	0

Table F.2 Seller Experiences in database

To test agents' dynamic negotiation capabilities, we assumed certain real time situations and certain data for participating agents in those assumed situations. The experiments are presented in the following subsections.

F.1 Experiment when agents dynamically enter and leave

Assume a situation where number of buyers and sellers want to buy or sell Printers. We consider the following buyers and sellers to dynamically enter or leave the market place. Note that italic lines are the output given by the application during the testing phase.

A seller named **Seller1 enters** the marketplace to sell a printer with the attitude of <0.6,0.8,0.6>, and warranty 2.0.

Seller1 retrieves the public price range from the Ontology and then retrieves the best relevant case from table SELLEREXP. **Seller1** uses the relevant case to slightly adjust the attitude and then calculates the negotiating price based on adjusted attitude, public price range and supply demand ratio as:
Change In Environment: Seller1 Max Price: 441.0 Min Price: 362.0
No Buyer for this Product yet... # of Sellers in market: 1.0
No need to notify agents anywhere

A buyer named **Buyer1 enters** the marketplace to buy a printer with the attitude of <0.7,0.8,0.5>, and warranty 2.0.

Buyer1 retrieves the public price range from the Ontology and then retrieves the best relevant case from table BUYEREXP. **Buyer1** uses the relevant case to slightly adjust the attitude and then calculates the negotiating price based on adjusted attitude, public price range and supply demand ratio as:
Change In Environment: Buyer1 Max Price: 397.5 Min Price: 312.0
Notify all agents buying or selling the product of new SDR from EW class...1.0
Change In Environment: Seller1 Max Price: 441.0 Min Price: 347.0

Seller1's clone and **Buyer1's** clone start negotiating for the printer from now on. Note that **Seller1** changes negotiating price range as the supply demand ratio changes drastically.

A seller named **Seller2 enters** the marketplace to sell a printer with the attitude of <0.5,0.9,0.8>, and warranty 2.0.

Seller2 retrieves the public price range from the Ontology and then retrieves the best relevant case from table SELLEREXP. **Seller2** uses the relevant case to slightly adjust the attitude and then calculates the negotiating price based on adjusted attitude, public price range and supply demand ratio as:
Change In Environment: Seller2 Max Price: 442.5 Min Price: 318.0

Environment Watcher at this point notifies the change in supply demand ratio to all the negotiating agents:
Notify all agents buying or selling the product of new SDR from EW class...2.0

Both **Seller1 and Buyer1** changes negotiating price range as the supply demand ratio changes drastically.
Change In Environment: Seller1 Max Price: 441.0 Min Price: 332.0
Change In Environment: Buyer1 Max Price: 375.0 Min Price: 312.0

Note that **Seller1** decreased minimum price from 341.0 to 332.0 and **Buyer1** decreased maximum price from 397.5 to 375.0 as the supply increased. Now **Buyer1** crates a clone for **Seller2** and **Seller2** creates a clone for **Buyer1** and they start negotiating from this point on. **Seller1's** clone and **Buyer1's** clone continue to negotiate without any interruption.

A buyer named **Buyer2 enters** the marketplace to buy a printer with the attitude of <0.3,0.8,0.4>, and warranty 2.0.

Buyer2 retrieves the public price range from the Ontology and then retrieves the best relevant case from table BUYEREXP. **Buyer2** uses the relevant case to slightly adjust the attitude and then calculates the negotiating price based on adjusted attitude, public price range and supply demand ratio as:
Change In Environment: Buyer2 Max Price: 387.0 Min Price: 306.0

Environment Watcher at this point notifies the change in supply demand ratio to all the negotiating agents:

Notify all agents buying or selling the product of new SDR from EW class...1.0

Seller1, Buyer1 and Seller2 changes negotiating price range as the supply demand ratio changes drastically.

Change In Environment: Seller1 Max Price: 441.0 Min Price: 347.0

Change In Environment: Buyer1 Max Price: 397.5 Min Price: 312.0

Change In Environment: Seller2 Max Price: 442.5 Min Price: 333.0

Note that **Seller1** increased minimum price from 332.0 to 347.0, **Buyer1** increased maximum price from 375.0 to 397.5 and Seller2 increased minimum price from 318.0 to 333.0 as the demand increased. Now **Buyer2** crates two clones, one for **Seller1** and the other for **Seller2. Seller1** and **Seller2** each in turn create a clone for **Buyer2** and they start negotiating. **Seller1's** clone and **Buyer1's** clone, **Seller2's** clone and **Buyer1's** clone continue to negotiate without any interruption.

A buyer named **Buyer3 enters** the marketplace to buy a printer with the attitude of <0.4,0.9,0.3>, and warranty 2.0.

Buyer3 retrieves the public price range from the Ontology and then retrieves the best relevant case from table BUYEREXP. **Buyer3** uses the relevant case to slightly adjust the attitude and then calculates the negotiating price based on adjusted attitude, public price range and supply demand ratio as:

Change In Environment: Buyer3 Max Price: 373.5 Min Price: 307.5

Environment Watcher at this point notifies the change in supply demand ratio to all the negotiating agents:

Notify all agents buying or selling the product of new SDR from EW class...0.6666666666666666

Seller1, Buyer1, Seller2 and Buyer2 changes negotiating price range as the supply demand ratio changes drastically.

Change In Environment: Seller1 Max Price: 441.0 Min Price: 352.0

Change In Environment: Buyer1 Max Price: 405.0 Min Price: 312.0

Change In Environment: Seller2 Max Price: 442.5 Min Price: 338.0

Change In Environment: Buyer2 Max Price: 394.5 Min Price: 306.0

Note that **Seller1** increased minimum price from 347.0 to 352.0, **Buyer1** increased maximum price from 397.5 to 405.0, **Seller2** increased minimum price from 333.0 to 338.0 and **Buyer2** increased maximum price from 387.0 to 394.0 as the demand increased. Now **Buyer3** crates two clones, one for **Seller1** and the other for **Seller2. Seller1** and **Seller2** each in turn create a clone for **Buyer3** and they start negotiating. **Seller1's** clone and **Buyer1's** clone, **Seller2's** clone and **Buyer1's** clone, **Seller1's** clone and **Buyer2's** clone and **Seller2's** clone and **Buyer2's** clone continue to negotiate without any interruption.

Buyer1 finishes negotiation with **Seller1** and **Seller2:**
Buyer1's Negotiation record with Seller1: 312.00/441.00/331.49/419.19/350.99/397.38

Buyer1's Negotiation record with Seller2: 312.00/442.50/331.49/412.56/350.99

Buyer2 finishes negotiation with **Seller1** and **Seller2:**
Buyer2's Negotiation record with Seller2: 306.00/442.50/323.33/416.17/340.67

Buyer2's Negotiation record with Seller1:

306.00/441.00/323.33/419.19/340.67/397.38/358.00

Buyer3 finishes negotiation with **Seller1** and **Seller2:**
Buyer3's Negotiation record with Seller2:

307.50/442.50/322.02/417.37/336.54/392.24/351.06

Buyer3's Negotiation record with Seller1:

307.50/441.00/322.02/420.35/336.54/399.70/351.06/379.06/365.58

Seller2 sold the product to Buyer3

Seller2 rejects Buyer1

Seller2 rejects Buyer2

match found 0.7647916666666666 updating CaseBase

Buyer3 Removes himself from the Market

match found 0.8154761904761905 updating CaseBase

Seller2 Leave the Marketplace

Environment Watcher removes **Seller2** and **Buyer3** from the marketplace and checks for the new supply demand ratio. Since the new supply demand ratio differs more than 0.1 from the old supply demand ratio, any agents who still didn't make a deal are notified. Note that there is no effect by this procedure in this situation. However, if the agents are still in the process of negotiation, agents will recalculate their negotiating price range.

Notify all agents buying or selling the product of new SDR from EW class...0.5

Change In Environment: Seller1 Max Price: 441.0 Min Price: 354.5

Change In Environment: Buyer1 Max Price: 408.75 Min Price: 312.0

Change In Environment: Buyer2 Max Price: 398.25 Min Price: 306.0

Seller1 sold the product to Buyer1

Seller1 rejects Buyer2

match found 0.85 updating CaseBase

Buyer1 Removes himself from the Market

Now the buyer agent experience record of a printer shows the following in the BUYEREXP table:

Pcat	Pname	War	PPR	Price	Attitude	SDR	NegRec	Res
CmpPro	printer	2	300-450	397.38	0.75,0.68,0.55	.583325	320/360/400	1

match found 0.8154761904761905 updating CaseBase

Seller1 Leave the Marketplace

Now the seller agent experience record of a printer shows the following in the SELLEREXP table:

Pcat	Pname	War	PPR	Price	Attitude	SDR	NegRec	Res
CmpPro	Printer	2	300-450	397.38	0.5,0.8,0.8	.583325	420/390/360	1

No Seller For this Product anymore...# of Buyers: 1.0

No deal from any Seller for this Buyer: Buyer2
Buyer2 Removes himself from the Market

No agents negotiating for this product anymore from EWNEW.....

From the above results we can clearly see that agents in this framework dynamically negotiate to compensate the changes that happens in the system.

F.2 Experiment when external changes occur

Assume a situation where number of buyers and sellers want to buy or sell a *tv*. We allow some buyers and sellers to dynamically enter or leave the market place, as we did in the previous experiment. Initially the public price range of a *tv* is set to 300 to 500 dollars. When the agents are negotiating in the system, we will change the public price range to 400-600 dollars and study the agents' behavior. Note that italic lines are the output given by the application during the testing phase.

A buyer named **Seller1 enters** the marketplace to buy a *tv* with the attitude of < 0.4,0.8,0.5>, and warranty 1.0.

Buyer1 retrieves the public price range from the Ontology and then retrieves the best relevant case from table BUYEREXP. **Buyer1** uses the relevant case to slightly adjust the attitude and then calculates the negotiating price based on adjusted attitude, public price range and supply demand ratio as:
Change In Environment: Buyer1 Max Price: 466.66666666666663 Min Price: 310.0
No Seller For this Product yet...# of Buyers: 1.0

A seller named **Seller1 enters** the marketplace to buy a *tv* with the attitude of < 0.8,0.6,0.7>, and warranty 1.0.

Seller1 retrieves the public price range from the Ontology and then retrieves the best relevant case from table SELLEREXP. **Seller1** uses the relevant case to slightly adjust the attitude and then calculates the negotiating price based on adjusted attitude, public price range and supply demand ratio as:

Change In Environment: Seller1 Max Price: 486.0 Min Price: 300.0
Notify all agents buying or selling the product of new SDR from EW class...1.0
Change In Environment: Buyer1 Max Price: 441.66666666666663 Min Price: 310.0

Buyer1's clone and **Seller1's** clone start negotiating for product from now on. Note that **Buyer1** changes negotiating price range as the supply demand ratio changes drastically.

Now we change the public price range of *tv* from 300 – 500 to 400 – 600:

CHANGE IN PPR FOR THE PRODUCT: tv
Change In Environment: Buyer1 Max Price: 530.0 Min Price: 410.0
Change In Environment: Seller1 Max Price: 586.0 Min Price: 400.0
As we can see above the negotiating agents' negotiating price ranges change drastically. Once Agents calculate their new price range they broadcast the new negotiation price range to their respective clones. Clones continue to negotiate for the product without any interruption.

A buyer named **Buyer2 enters** the marketplace to buy a *tv* with the attitude of < 0.7,0.5,0.8>, and warranty 1.0.

Buyer2 retrieves the public price range from the Ontology and then retrieves the best relevant case from table BUYEREXP. **Buyer2** uses the relevant case to slightly adjust the attitude and then calculates the negotiating price based on adjusted attitude, public price range and supply demand ratio as:

Change In Environment: Buyer2 Max Price: 599.0 Min Price: 412.0

Environment Watcher at this point notifies the change in supply demand ratio to all the negotiating agents:

Notify all agents buying or selling the product of new SDR from EW class...0.5

Both **Seller1 and Buyer1** changes negotiating price range as the supply demand ratio changes drastically.

Change In Environment: Buyer1 Max Price: 545.0 Min Price: 410.0

Change In Environment: Seller1 Max Price: 586.0 Min Price: 410.0

Again clones of **Seller1** and **Buyer1** continue to negotiate without any interruption.

From the results bellow we can see that **Buyer1** and **Seller1** finished their negotiation just before the public price for *tv* has been changed. As a result, **Seller1** accepted **Buyer1's** 310 dollars bid. However, when the **Seller1** finished its negotiation with **Buyer2**, it chooses to sell the product to **Buyer2** as **Buyer2** is willing to pay more for the *tv*.

Buyer1's Negotiation record with Seller1: 310.00

Buyer2's Negotiation record with Seller1: 412.00

Seller1 sold the product to Buyer2

match found 0.7229166666666668 updating CaseBase

Buyer2 Removes himself from the Market

Now the buyer agent experience record of a *tv* shows the following in the BUYEREXP table:

Pcat	Pname	War	PPR	Price	Attitude	SDR	NegRec	Res
homeelec	tv	1	400-600	412	0.65,0.45,0.75	.75	412.00	1

Seller1 rejects Buyer1

match found 0.7375 updating CaseBase

Seller1 Leave the Marketplace

Now the seller agent experience record of a *tv* shows the following in the SELLEREXP table:

Pcat	**Pname**	**War**	**PPR**	**Price**	**Attitude**	**SDR**	**NegRec**	**Res**
homeelec	tv	1	400-600	412	0.6,0.5,0.6	.75	412.00	1

No Seller For this Product anymore...# of Buyers: 1.0

No deal from any Seller for this Buyer: Buyer1

Buyer1 Removes himself from the Market

No agents negotiating for this product anymore from EWNEW.....

From the above results, it is clear that agents in this framework can dynamically negotiate and compensate for the change that happens outside the e-commerce environment.

Appendix G

Documented Code

All the classes and database schema for this application are shown bellow.

```
/**************************************
This class EWNEW serves as the main class where we input the information of buyers and
sellers for a product. It is the controller/mediator of the entire system.
**************************************/

import java.util.*;
import java.sql.*;
import java.io.*;
import java.lang.*;

public class EWNEW
{
  static BSLNEW bslnew = new BSLNEW();
  Case C = new Case();
  public static void main (String[] args){
   EWNEW test = new EWNEW();
   test.AddBuyerSeller("Buyer1", "tv", "buyer", "0.4,0.8,0.5", "1.0", "315", "200");
   test.AddBuyerSeller("Seller1", "tv", "seller", "0.8,0.6,0.7", "1.0", "500", "315");
   test.PublicPriceRangeChangeForProduct("tv", 600.00, 400.00);
   test.AddBuyerSeller("Buyer2", "tv", "buyer", "0.7,0.5,0.8", "1.0", "500", "315");
}//end of main

//This method enables collecting the references of all agents (buyers and //sellers)for a
//certain product and notify them of the change in Public Price Range

public void PublicPriceRangeChangeForProduct(String Pname, double pprMax, double pprMin){
  System.out.println("CHANGE IN PPR FOR THE PRODUCT : " + Pname);
  Vector VecBuySell = new Vector(10);
  C.UpdatePPRinCaseBase(Pname, pprMax, pprMin);
  VecBuySell = bslnew.WhoAreTheBuyersSellers(Pname);
   for(int i = 0; i< VecBuySell.size(); i++){
      ActualAgent AA = (ActualAgent)VecBuySell.get(i);
      AA = (ActualAgent)VecBuySell.get(i);
      AA.Notify(VecBuySell, pprMax, pprMin);
   }
  }// End of class PublicPriceRangeChangeForProduct
```

```
//This method enables adding a buyer or seller to the dynamic BuyerSellerList and to collect
//references of buyers and sellers for a certain product after adding the new buyer or
//seller. Then it notifies the agents of the change that happened in the Marketplace
//Environment.

public void AddBuyerSeller(String AName, String PName, String BuySell, String Attitude,
String Warranty, String MaxPrice, String MinPrice){
  Vector VecBuySell = new Vector(20);
  VecBuySell.clear();
  ActualAgent AA = null;
  double NumBuyer = 0, NumSeller = 0;
  double SDR = -1.0;
  String AgentNameExist="";
  VecBuySell = bslnew.AddBuyerSeller(AName, PName, BuySell, Attitude, Warranty, MaxPrice,
  MinPrice);
  if(VecBuySell.size()==1){
    Object ob = VecBuySell.elementAt(0);
    if(ob instanceof String)AgentNameExist = (String)ob;
      if( (AgentNameExist).equals("Exists") ){
         System.out.println("Agent name already Exists in B/S list...");
         System.out.println("Notify the interface that given agent Will not be
         created becase agent name already exist in the system...");
      }
      else{
           AA = (ActualAgent)VecBuySell.elementAt(0);
           if((AA.BS).equals("buyer"))NumBuyer++;
           else NumSeller++;
           try{
               if(!(NumBuyer == 0)) SDR = (double)(NumSeller/NumBuyer);
               else{
                    System.out.println("No Buyer for this Product yet... " + "#
                    of Sellers in market: " + NumSeller+ " Or send SDR = 0");
                    System.out.println("No need to notify the agents anywhere");
               }
           }
           catch (ArithmeticException e){
                 System.out.println("No Buyer for this Product yet... " + "# of Sellers in
                 market: " + NumSeller+ " Or send SDR = 0");
           }
           if(SDR==0) System.out.println("No Seller For this Product yet..."+ "# of Buyers:"
           + NumBuyer + " SDR = " + SDR);
           else System.out.println("SDR = 0 - only one Buyer or Seller...");
     }//else ends
   }//if ends
```

```
else{
     Enumeration enu = VecBuySell.elements();
     while(enu.hasMoreElements()){
          AA =(ActualAgent)enu.nextElement();
          if((AA.BS).equals("buyer"))NumBuyer++;
          else NumSeller++;
     }//whlie ends
     try{
         if(!(NumBuyer == 0))SDR = (double)(NumSeller/NumBuyer);
     }
     catch (ArithmeticException e){
           System.out.println("Devided by zero at EW class in agent adding method..");
     }
     if(NumBuyer == 0)System.out.println("No Buyer for this Product yet..."+ "# of Sellers
         in market: " + NumSeller+ " Or send SDR = 0");
       if(SDR==0) System.out.println("No Seller For this Product yet..."+ "# of Buyers:"
          +NumBuyer+ "seller:" + NumSeller+" SDR = " + SDR);
       else if(SDR > 0){
        System.out.println("Nortify all agents buying or selling the product of new SDR
         from EW class..." + SDR);
        for(int i = 0; i< VecBuySell.size(); i++){
           AA = (ActualAgent)VecBuySell.get(i);
           AA.Notify(SDR,VecBuySell);
        }
        for(int i = 0; i< VecBuySell.size(); i++){
          AA = (ActualAgent)VecBuySell.get(i);
          AA.startNegotiation();
        }
      }
  }//end of else
}//End of method AddBuyerSellerAgent

//When an Agent (Buyer or Seller) finishes a deal or gets rejected by all the opponents we
//want to remove them from the buyer seller list. Since It will be always the same I have
//given a Static class ForAgents and the RomveBuyerSeller Method inside the static class!!
//This enables me to freely have access to EnvironemtWatcher class(this) and yet not to pass
//reference of this main class to all the buyers and sellers.

public static class ForAgents
{
//1.This method RemoveAnAgent is responsible for Removing a buying or selling agent from the
//BuyerSellerList
```

```
//2.Nortifying all the buying and selling agents of the same product regarding the changes
//in the Environment because the removal of the agent ie, Change of SDR to all buying and
//selling agents.

public void RemoveBuyerSeller(String PName, String AName, String BuySell)
{
  Vector VecBuySell = new Vector(20);
  VecBuySell.clear();
  ActualAgent AA;
  double NumBuyer = 0, NumSeller = 0;
  double SDR = -1.0;
  String NoAgentExist="";
  VecBuySell = bslnew.RemoveBuyerSeller(PName, AName, BuySell);
   if(VecBuySell.size()==1){
     Object ob = VecBuySell.elementAt(0);
     if(ob instanceof String)NoAgentExist = (String)ob;
       if( (NoAgentExist).equals("None") )
       {System.out.println("No agents negotiating for this product anymore from
          EWNEW.....");
       }
       else{
            AA = (ActualAgent)VecBuySell.elementAt(0);
            if((AA.BS).equals("buyer"))NumBuyer++;
            else NumSeller++;
            try{
                if(!(NumBuyer == 0)) SDR = (double)(NumSeller/NumBuyer);
            else{
                System.out.println("No Buyer for this Product Anymore... " + "# of Sellers
                    in market: " + NumSeller+ " Or send SDR = 0");
                System.out.println("No need to notify any agents anywhere");
            }
            }
            catch (ArithmeticException e){
            System.out.println("No Buyer for this Product yet... " + "# of Sellers in
             market: " + NumSeller+ " Or send SDR = 0");
            }
         if(SDR==0) System.out.println("No Seller For this Product anymore..."+ "# of
           Buyers:" + NumBuyer + " SDR = " + SDR);
         else System.out.println("SDR = 0 - only one Buyer or Seller...");
       }//else ends
     }//if ends
     else{
       Enumeration enu = VecBuySell.elements();
       while(enu.hasMoreElements()){
```

```
        AA =(ActualAgent)enu.nextElement();
        if((AA.BS).equals("buyer"))NumBuyer++;
        else NumSeller++;
      }//whlie ends
      try{
          if(!(NumBuyer == 0))SDR = (double)(NumSeller/NumBuyer);
      }
      catch (ArithmeticException e){
         System.out.println("Devided by zero at EW class in agent adding method..");
      }
      if(NumBuyer == 0)System.out.println("No Buyer for this Product Anymore..."+ "# of
         Sellers in market: " + NumSeller+ " Or send SDR = 0");
      if(SDR==0) System.out.println("No Seller For this Product Anymore..."+ "# of Buyers:"
         +NumBuyer+ "seller:" + NumSeller+" SDR = " + SDR);
      else if(SDR > 0){
             System.out.println("Nortify all agents buying or selling the product of new
             SDR from EW class..." + SDR);
            for(int i = 0; i< VecBuySell.size(); i++){
               AA = (ActualAgent)VecBuySell.get(i);
               AA.Notify(SDR,VecBuySell);
            }
           for(int i = 0; i< VecBuySell.size(); i++){
              AA = (ActualAgent)VecBuySell.get(i);
              AA.startNegotiation();
           }
      }
    }//end of else
  }//end of method RemoveBuyerSeller
 }//END OF STATIC CLASS FOR-AGENTS
}//End of class EWNEW


/**************************************
This class is used to model the DYNAMIC BUYER SELLER LIST where information about the
AGENT NAME and the reference to all agents are KEPT UNIQUELY IN ORDER TO ASSIST ENVIRONMENT
WATCHER.
**************************************/

import java.util.*;
import java.sql.*;
import java.net.URL;
import java.io.*;

public class BSLNEW
```

```
{
 Vector API = new Vector(100);
 Vector VecBuySell = new Vector(20);
 //AddBuyerSeller is responsible for adding new agent (buyer or seller) to the dynamic list
 //of buyers and sellers, ie API Vector. Also it is responsible for returning list of
 //buyers and seller for the product in question to the EnvironmentWatcher so that
 //EvnvironmentWatcher can calculate the NEW SDR for the product. ** If an agent name is
 //already found a fake object will be send to Environment where the AgentName of the
 //object will contatin "Exists" string to indicate agent object will not be created with
 //the given agent name.*/

 public Vector AddBuyerSeller(String AName, String PName, String BuySell, String Attitude,
                                    String Warranty, String MaxPrice, String MinPrice){
   boolean AnameExists = false;
   VecBuySell.clear();
   ActualAgent AA;
   double NumBuyer = 0, NumSeller = 0;
   double sdr = 0.0;
   String SDR = "";
    if(API.isEmpty()){
     SDR = "0.0";
     if(BuySell.equals("buyer"))
       AA = new BuyerAgent(AName, PName, BuySell, Attitude, Warranty, MaxPrice, MinPrice,
                                                                        SDR, VecBuySell);
     else AA = new SellerAgent(AName, PName, BuySell, Attitude, Warranty, MaxPrice,
                                                              MinPrice, SDR, VecBuySell);
     API.addElement(AA);
     VecBuySell.addElement(AA);
     return VecBuySell;
    }//if over
    else{
         Enumeration enum = API.elements();
        while(enum.hasMoreElements()){//For each buyer or seller in the list
           AA =(ActualAgent)enum.nextElement();
           if(!(AA.AN).equals(AName)){
             if((AA.PN).equals(PName)){
                VecBuySell.addElement(AA);
                 if((AA.BS).equals("buyer"))NumBuyer++;
                 else NumSeller++;
             }
           }
           else if((AA.AN).equals(AName)){
                  AnameExists = true;
           }//else if inside while over
```

```
      }//while over
     }//Main else over
  if(AnameExists == false){
   if(BuySell.equals("buyer"))NumBuyer++;
   else NumSeller++;
   if(NumBuyer==0)SDR = "0.0";
   if(NumSeller==0)SDR = "0.0";
   if(!(NumBuyer==0) &&!(NumSeller==0))
     sdr =  (double)(NumSeller/NumBuyer);
     SDR = Double.toString(sdr);
     if(BuySell.equals("buyer"))AA = new BuyerAgent(AName, PName, BuySell, Attitude,
                                  Warranty, MaxPrice, MinPrice, SDR, VecBuySell);
     else AA = new SellerAgent(AName, PName, BuySell, Attitude, Warranty, MaxPrice,
                                                        MinPrice, SDR, VecBuySell);
     VecBuySell.addElement(AA);
     API.addElement(AA);
   }//if over
   else {
         VecBuySell.clear();
         String AgentExists = new String("Exists");
         VecBuySell.addElement(AgentExists);
   }
  return VecBuySell;
 }//Method AddBuyerSeller is over


//RemoveBuyerSeller is responsible for removing an agent (buyer or seller) from the dynamic
//list of buyers and sellers, ie API Vector. Also it is responsible for returning list of
//buyers and seller for the product in question to the EnvironmentWatcher so that
//EvnvironmentWatcher can calculate the NEW SDR for the product. ** If an agent name is
//doesn't exists, a fake object will be send to EnvironmentWatcher where the AgentName of
//the object will contain "None" string to indicate that agent doesn't exist in List of
//Buyers and Sellers.

public Vector RemoveBuyerSeller(String productName, String AName, String buyerseller)
{
  VecBuySell.clear();
  boolean PnameExists = false;
  ActualAgent AA;
   if(API.isEmpty()){
      String AgentExists = new String("None");
      VecBuySell.addElement(AgentExists);
      return VecBuySell;
   }//if over
```

```
  else{
       Enumeration enum = API.elements();
       int removeIndex = -1;
       for(int i=0; i<API.size(); i++){//For each buyer or seller in the list
          AA = (ActualAgent)API.elementAt(i);
          if((AA.AN).equals(AName)&&(AA.PN).equals(productName)){
            removeIndex = i;
          }
          else if((AA.PN).equals(productName)){
             PnameExists = true;
             VecBuySell.addElement(AA);
          }//else if inside while over
        }//for is over
       if(!(removeIndex==-1))API.remove(removeIndex);
  }//Main else over
  if(PnameExists==false){
     String AgentExists = new String("None");
     VecBuySell.addElement(AgentExists);
  }
  return VecBuySell;
}//end of mehtod RemoveBuyerSeller

//WhoAreTheBuyersSellers is responsible returning all the agents' who are involved in buying
//or selling a certain product.

public Vector WhoAreTheBuyersSellers(String Pname){
    Vector VecBuySell = new Vector(10);
    for(int i=0; i<API.size(); i++){//For each buyer or seller in the list
       ActualAgent AA = (ActualAgent)API.get(i);
        if(Pname.equals(AA.PN))
           VecBuySell.addElement(AA);
        }
        return VecBuySell;
   }
}//end of class BSLNEW


/**************************************
This class is used to model the ontology where information about the products is returned
from the database, using JDBC. Table "range" contains PPR for each product and table
"simvalue" contains the similarity between product categories
**************************************/

import java.util.*;
```

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

public class Ontology
{
 //The product similarity computed here
  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 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); //To return the similar Products
                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);
                       Statement stmt = conn.createStatement();
                       //To find the avearge price pr1 of Price Similarity
                       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;
//Found pr1 for Price Similarity function
Averpriceactu =(double)( Pmaxactu + Pminactu )/(double)2.0;
//To find the conceptually similar products
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);
}//Conceptually similar products found.
Enumeration simcat = Simcats.elements();
//To find avearage price pr2 of PriceSimilarity
//function needed a Concepsim object
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 ;
        //pr2 for PS function is found
        Avgpriceother = (Pmaxother + Pminother)/(double)2.0;
         Match = productSimilarity (prosim.simvalue,Averpriceactu,
                                          Avgpriceother);
        double threshold = (double)0.5;//0.75
        if (Match >= threshold)
```

```
                        {
                        Simproduct similarpro = new Simproduct();
                        //Place the similar product info in object similarpro
                        similarpro.Pname = Prodnameother;
                        similarpro.Pcat = Prodcat1;
                        similarpro.Pmax = Pmaxother;
                        similarpro.Pmin = Pminother;
                        similarpro.PSim = Match;
                        //Similar Products in a Vector to return
                        Similarproducts.addElement(similarpro);
                        }
                    }
                   }
                   conn.close();
              }catch (SQLException sqlex){
                    System.err.println("Failed to load JDBC/ODBC driver.");
                    sqlex.printStackTrace();
              }
        return  Similarproducts;
}

//This function 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);
            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;
   }
}// End of calss Ontology


/************************************
This is a data structure used by Ontology class to keep the information about two product's
Conceptual Similarity. The product's conceptual similarity Catogory and their similarity
value is kept here to be used by ReturnSimilarity function in Ontology class.
************************************/

import java.io.*;
import java.util.*;

public class Concepsim
{
    public String prodcat1;//One product's Product Catogory
    public String prodcat2;//Other product's Product Catogory
    public double simvalue;//Similarity between above two
}//end of class Concepsim


/****************************************
This class's responsibility is to find the best similar case from the case base for a given
situation.  It uses class ontology to get the similar products first and then looks for
products in the case base that are also similar in Attitude and PPR so that it can return
similar relevant experience. In the similar fashon this class is also responsible for
updating or adding negotiation records to the Case base database.
****************************************/
```

```
import java.sql.*;
import java.net.URL;
import java.io.*;
import java.util.*;

public class Case
{
 Ontology onto;
 Product prod;
 double ImpTimePast, ImpPricePast, CommitPast, PricePast;
 double PricePresneg;
 double PricePres;
 double Pmaxactu, Pminactu;
 double Pmaxother, Pminother;
 double WarrPast=0.0, WarrPres;
 double SdrPast;
 double ProductSim, RangeSim, AttitudeSim, NegotiationSim;
 double NegSim;
 double Match;
 double MaxMatch =(double) 0.7;
 double NegMatch = (double) 0.5;
 String Pname, Pcat;
 String NegRec, saveNegotiation;
 String Bestcase;
 Vector SimProducts;

 public Case()
 {
  onto = new Ontology();
 }//Ref. to Ontology obj.

 //This method find the prduct in question's(as actual) details
 public void getProductDetails(String Pname)
 {
  String tester;
  prod = onto.getProdDet(Pname);
  Pmaxactu = prod.Pmax;
  Pminactu = prod.Pmin;
  tester = prod.Pname;
 }

 public double getPmaxactu()
 {
```

```
  return Pmaxactu;
}
public double getPminactu()
{
  return  Pminactu;
}

//Method bellow 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);//Vector holding ProductSimilarity Objects
  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);
      Statement stmt = conn.createStatement();
      while(enum.hasMoreElements())//For each product that is similar from Vector do bellow
      {
       Simproduct similarpro;
       similarpro = (Simproduct) enum.nextElement();
       String PnameOther;
       PnameOther = similarpro.Pname;
       Pmaxother = similarpro.Pmax;
       Pminother = similarpro.Pmin;
       ProductSim = similarpro.PSim;

       //if the agent is seller
       String statement  = "Select * from sellerexp where PRODNAME ='"+PnameOther+"'";
       //if the agent is buyer
       if(User.equalsIgnoreCase("buyer"))
       statement = "Select * from buyerexp where PRODNAME ='"+PnameOther+"'";
       ResultSet rs = stmt.executeQuery(statement );
```

```
String Attitudeother, Pastneg, othercase, Pcatother;
double Pricepast, SdrPast, WarrPast;
int PastRes;//Was negotiation successful or not
     while(rs.next())//for each product experience form CaseBase do the bellow
      {
           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());
             }
           getProductDetails(Pname
           othercase = (Pcatother + "|" + PnameOther + "|" + WarrPast + "|" +
                        Pmaxother + "|" + Pminother + "|" + Pricepast + "|" +
                        Attitudeother+ "|" + SdrPast + "|" + Pastneg + "|" +
                        PastRes + "|" + Pmaxactu + "|" + Pminactu);
           //Finding Attitude Similarity
           AttitudeSim = attitudeSimilarity(ImpTimePast,ImpTimePres, ImpPricePast,
                                            ImpPricePres,CommitPast,CommitPres);
           //Finding Public Price Range Similarity
           RangeSim = rangeSimilarity(Pmaxactu,Pminactu,Pmaxother,Pminother);
           //Relevant Experience mach value
           Match = ((AttitudeSim + RangeSim + ProductSim)/(double)3.0);
           if( Match >= MaxMatch)
            {
              Bestcase = othercase;
              MaxMatch = Match;
            }
       }//For each product experience from case base (while loop) ends
  }//For each product that is similar from Vector (while loop) ends
  conn.close();
}//End of try
catch (SQLException sqlex)
```

```
  {
    System.err.println("Failed to load JDBC/ODBC driver!!!!!.");
    sqlex.printStackTrace();
  }
 return Bestcase;
}//End of Method getBestCase

 //This functions returns Attitude similarity

 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 functions returns similarity of two public price ranges

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;
 }

 //Method bellow updates or adds a case to the Negotiation Record either to buyerexp or
 //sellerexp table.
 public void UpdateOrAddNegotiation(String matcher){
   String User = "", Pnameother="", attPres="";
   int resPres = 0;
   double ImpTimePres = 0, ImpPricePres = 0, CommitPres = 0, SdrPres = 0;
   boolean negRecMatch = false;
   //tokenize the string
   StringTokenizer tokens = new StringTokenizer(matcher,"|");
   while(tokens.hasMoreTokens())
   {
    User = tokens.nextToken();
    Pcat = tokens.nextToken();
    Pname = tokens.nextToken();
    WarrPres = Double.parseDouble(tokens.nextToken());
    Pmaxactu = Double.parseDouble(tokens.nextToken());
    Pminactu = Double.parseDouble(tokens.nextToken());
```

```
  PricePres = Double.parseDouble(tokens.nextToken());
  attPres = tokens.nextToken();
  SdrPres = Double.parseDouble(tokens.nextToken());
  NegRec = tokens.nextToken();
  resPres = Integer.parseInt(tokens.nextToken());
  StringTokenizer tok = new StringTokenizer(attPres,",");
    while(tok.hasMoreTokens())
    {
     ImpTimePres= Double.parseDouble(tok.nextToken());
     ImpPricePres= Double.parseDouble(tok.nextToken());
     CommitPres = Double.parseDouble(tok.nextToken());
    }
 }//end of while (string tokenizing)
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);
    Statement stmt = conn.createStatement();
    //if the agent is seller
    String statement  = "Select * from sellerexp where PRODNAME ='"+Pname+"'";
    //if the agent is buyer
    if(User.equalsIgnoreCase("buyer"))
    statement = "Select * from buyerexp where PRODNAME ='"+Pname+"'";
    ResultSet rs = stmt.executeQuery(statement);
     String Attitudeother="", Pastneg, othercase, Pcatother;
     double Pricepast=0, SdrPast=0, WarrPast=0;
     int PastRes;//Was negotiation successful or not
          while(rs.next())//for each product experience form CaseBase do the bellow
           {
                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 toke = new StringTokenizer(Attitudeother,",");
  while(tokens.hasMoreTokens())
  {
   ImpTimePast= Double.parseDouble(toke.nextToken());
   ImpPricePast= Double.parseDouble(toke.nextToken());
   CommitPast = Double.parseDouble(toke.nextToken());
  }
 othercase = (Pcatother + "|" + Pnameother + "|" + WarrPast + "|" +
              Pmaxother + "|" + Pminother  + "|" + Pricepast + "|" +
              Attitudeother+ "|" + SdrPast + "|" +Pastneg + "|"
              + PastRes );


 if(Pname.equals(Pnameother)/* && (Pmaxactu == Pmaxother)&& (Pminactu ==
                                                    Pminother)*/){
//Finding Product Similarity (CS is always 1 here since products are same)
    double price1 = (double)(Pmaxactu + Pminactu) / (double)2.0;
    double price2 = (double)(Pmaxother + Pminother) / (double)2.0;
    ProductSim = 1.0 * (1- Math.abs(price1-price2)/Math.max
                                                    (price1,price2));
    //Finding Attitude Similarity
    AttitudeSim = attitudeSimilarity(ImpTimePast,ImpTimePres,ImpPricePast,
                                    ImpPricePres,CommitPast,CommitPres);
    //Finding Public Price Range Similarity
    RangeSim = rangeSimilarity(Pmaxactu,Pminactu,Pmaxother,Pminother);
    //Finding NegotiationSimilarity
    NegotiationSim = NegotiationSimilarity(Pastneg, NegRec, PastRes,
                                                        resPres);
    //Negotiation Record Match value
    Match = ((AttitudeSim + RangeSim + ProductSim +
                                            NegotiationSim)/(double)4.0);
    if( Match >= NegMatch)
     {
       negRecMatch = true;
       Bestcase = othercase;
       NegMatch = Match;  System.out.println("match found " + Match);
     }
  }//End of if
}//end of rs.next's while
```

```
//NOW USE THE othercase TO MERGE NEGOTIATION IN CBR SYSTEM(IFGOODMATCHsFOUND)
//OR IF othercase IS NULL THEN ADD A NEW CASE TO CBR SYSTEM.
if(negRecMatch){
  WarrPres = (WarrPast + WarrPres)/(double)2.0;
  PricePres = FindLastPriceToUpdateCaseBase(NegRec);
  attPres = AverageAttitude(attPres, Attitudeother);
  SdrPres = ((SdrPres + SdrPast)/(double)2.0);
  NegRec = saveNegotiation;
   String Statement="";
   if(User.equalsIgnoreCase("buyer")){
    Statement = "UPDATE buyerexp SET WARRANTY = '"+WarrPres+"', PPRMAX
      ='"+Pmaxactu+"', PPRMIN ='"+Pminactu+"', PRICE = '"+PricePres+"',
      ATTITUDE = '"+attPres+"', SDR = '"+SdrPres+"', NEGREC = '"+NegRec+"'
      WHERE PRODNAME = '"+Pname+"'";
     stmt.execute(Statement);
    //System.out.println(Statement);
   }
   else{
    Statement = "UPDATE sellerexp SET WARRANTY = '"+WarrPres+"', PPRMAX
      ='"+Pmaxactu+"', PPRMIN ='"+Pminactu+"',PRICE = '"+PricePres+"',
      ATTITUDE = '"+attPres+"', SDR = '"+SdrPres+"', NEGREC = '"+NegRec+"'
      WHERE PRODNAME = '"+Pname+"'";
     stmt.execute(Statement);
    //System.out.println(" Updating .." + Statement);
   }
}
else if(Pnameother.equals(Pname)) {
     String Statement="";
       if(User.equalsIgnoreCase("buyer")){
          Statement = "INSERT INTO buyerexp VALUES ('"+Pcat+"', '"+Pname+"',
             '"+WarrPres+"', '"+Pmaxactu+"', '"+Pminactu+"', '"+PricePres+"',
             '"+attPres+"','"+SdrPres+"', '"+NegRec+"', '"+resPres+"')";
          //System.out.println(Statement);
          stmt.execute(Statement);
       }
       else{
          Statement = "INSERT INTO sellerexp VALUES ('"+Pcat+"', '"+Pname+"',
             '"+WarrPres+"', '"+Pmaxactu+"', '"+Pminactu+"', '"+PricePres+"',
             '"+attPres+"','"+SdrPres+"', '"+NegRec+"', '"+resPres+"')";
          stmt.execute(Statement );
          //System.out.println(Statement);
       }
}
//if products are same... then add a new case (just to be safe)!!
```

```
     conn.close();
    }//End of try
    catch (SQLException sqlex)
    {
     System.err.println("Failed to load JDBC/ODBC driver!!!!!.");
     sqlex.printStackTrace();
    }

}//End of method UpdateOrAddNegotiation

//This functions returns Negotiation similarity of two Negotiations: both successful and
//failure cases!!

public double NegotiationSimilarity(String Pastneg, String NegRec, int PastRes,
                                                                    int resPres){
 double negsim = 0;
 int counterPast = 0, counterPres = 0;
 StringTokenizer past = new StringTokenizer(Pastneg,"/");
 StringTokenizer pres = new StringTokenizer(NegRec,"/");
   while(past.hasMoreTokens())
    {
      past.nextToken();counterPast++;
    }
    while(pres.hasMoreTokens())
    {
      pres.nextToken();counterPres++;
    }
    if(counterPres>=counterPast)saveNegotiation = Pastneg;
    else saveNegotiation = NegRec;
    if(PastRes == resPres){
      negsim = 1 - ((double)Math.abs(counterPast - counterPres)/(double)Math.max
                                                    (counterPast,counterPres));
    }
    else{
         negsim = 1 - ( ( (double)Math.abs(counterPast - counterPres)/(double)Math.max
                                          (counterPast,counterPres)) * 0.5 + 0.5);
    }
  return negsim;
 }

//This functions returns avarage of the attitude (IT, IP, C)
public String AverageAttitude(String attPres, String Attitudeother){
   String aveAttitude="";
   StringTokenizer AP = new StringTokenizer(attPres,",");
```

```
  StringTokenizer AO = new StringTokenizer(Attitudeother,",");
  double ImpTimeP=0,ImpPriceP=0, CommitP=0, ImpTimeO=0, ImpPriceO=0, CommitO=0;
  while(AP.hasMoreTokens()){
     ImpTimeP= Double.parseDouble(AP.nextToken());
     ImpPriceP= Double.parseDouble(AP.nextToken());
     CommitP = Double.parseDouble(AP.nextToken());
  }
  while(AO.hasMoreTokens()){
     ImpTimeO= Double.parseDouble(AO.nextToken());
     ImpPriceO = Double.parseDouble(AO.nextToken());
     CommitO = Double.parseDouble(AO.nextToken());
  }
  aveAttitude = Rounding.toString(((ImpTimeP+ImpTimeO)/(double)2.0),2)+ ",";
  aveAttitude += Rounding.toString(((ImpPriceP+ImpPriceO)/(double)2.0),2)+",";
  aveAttitude += Rounding.toString(((CommitP+CommitO)/(double)2.0),2);

 return aveAttitude;
}

//This functions returns the price of the successful or unsuccessful deal For buyers: its
//buyer's and for sellers: its seller's last price.

public double FindLastPriceToUpdateCaseBase(String NegRec)
{
  double price = 0;
  StringTokenizer pres = new StringTokenizer(NegRec,"/");
     while(pres.hasMoreTokens())
     {
       price = Double.parseDouble(pres.nextToken());
     }
return price;
}

public void UpdatePPRinCaseBase(String Pname, double pprMax, double pprMin){
  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);

        Statement stmt = conn.createStatement();
        String Statement = "UPDATE range SET MINPRICE = '"+pprMin+"', MAXPRICE = '"+pprMax+"'
                                                    WHERE PRODNAME = '"+Pname+"'";
        stmt.execute(Statement);
     conn.close();
    }
    catch (SQLException sqlex)
    {
        System.err.println("Failed to load JDBC/ODBC driver!!!!!.");
        sqlex.printStackTrace();
    }
}//End of Method UpdatePPRinCaseBase
}//End of class Case.java
/**********************************
Product information is a data structure used for temporarlay storing and
manupulating/identifying the price ranges and it is used by Case class to find the public
Maximum and Minimum price ranges for a given product.
**********************************/

public class Product
{
    public String Pcat; //Product Catagory form SIMVALUE table (DB)
    public String Pname;//Product Name
    public double Pmin; //Priduct public minimum price
    public double Pmax; //Product public maximum price

    public Product()
    {
        Pname = "";
        Pcat = "";
        Pmax = 0;
        Pmin = 0;
    }
}// End of class Product

/**************************************
This class is used to round the double numbers to whatever the precision we want so that we
can save the values in the Case Based Reasoning System.
**************************************/
```

```
import java.lang.Math;

public class Rounding
{
  public static String toString (double d, int place)
  {
    if (place <= 0)
      return ""+(int)(d+((d > 0)? 0.5 : -0.5));
    String s = "";
    if (d < 0)
      {
       s += "-";
       d = -d;
      }
    d += 0.5*Math.pow(10,-place);
    if (d > 1)
      {
       int i = (int)d;
       s += i;
       d -= i;
      }
    else
      s += "0";
    if (d > 0)
      {
       d += 1.0;
       String f = ""+(int)(d*Math.pow(10,place));
       s += "."+f.substring(1);
      }
    return s;
  }
 //Rounding.toString(d,4)
}//End of class Rounding


/****************************************
This is a data structure used by Case class. The similar products found is kept in instance
of this class temporarlay and return these information to CaseBase during Product
Similarity.
****************************************/

public class Simproduct
{
    public String Pname;//Product Name
```

```
    public String Pcat; //Product Catagory form SIMVALUE table (DB)
    public double Pmax; //Product public maximum price
    public double Pmin; //Priduct public minimum price
    public double PSim; //The similarity value found for this product and the product in
                                                                                question.
}//End of Class Simproduct


/**********************************
Agent is an abstract class which extends to Thread and it has the most common attibutes and
the functionalities of all the Buyer Agents, Seller Agents, Buyer Clones and Seller Clones.
**********************************/

import java.util.*;

public abstract class Agent extends Thread
{
 String AN,PN,BS,ATT,AW,MAX,MIN,SDR;
 double aip = 0.0, ait= 0.0, amaxp = 0.0, aminp = 0.0;
 Vector LBS;

 //Constructor for ActualAgent
 public Agent(String AN, String PN, String BS, String ATT, String AW, String MAX, String
                                                        MIN, String SDR, Vector LBS){
    this.AN = AN;
    this.PN = PN;
    this.BS = BS;
    this.ATT=ATT;
    this.AW = AW;
    this.MAX=MAX;
    this.MIN=MIN;
    this.SDR=SDR;
    this.LBS=LBS;
 }

 //Constructor for CloneAgent
 public Agent(double aip, double ait, double amaxp, double aminp){
   this.aip   = aip;
   this.ait   = ait;
   this.amaxp = amaxp;
   this.aminp = aminp;
 }
  public String getAgentName(){
    return AN;
```

```
  }

  //BuyerAgent, SellerAgent, BuyerClone or SellerClone Notification
  public abstract void Notify(double sdr,Vector listBs);
  //Agents to start Negotiation (where run is kept)
  public void startNegotiation(){
  }
}// End of abstract class Agent


/************************************
Abstract class ActualAgent is a subclass of abstract class Agent. Here we have the most
common attributes and methods of BuyerAgent or Seller Agent. As we need to have Threads for
Buyer and Seller Agents we implement Runnable here
************************************/

import java.util.*;

public abstract class ActualAgent extends Agent implements Runnable
{
  //To CaseBaseSystem Component
  Case C = new Case();
  EWNEW EwNew = new EWNEW();
  EWNEW.ForAgents FA = EwNew.new ForAgents();
  double IT =0.0, IP=0.0, IC=0.0, ASDR = 0.0, AW = 0.0;
  double UserMax = 0.0, UserMin = 0.0;
  double BCWarr=0.0, BCPMax=0.0, BCPMin=0.0, BCPrice=0.0, BCSDR=0.0;
  double ThisProductsPPRMax=0.0, ThisProductsPPRMin=0.0, BCIT=0.0, BCIP=0.0, BCIC=0.0;
  double AdjustedIT = -1, AdjustedIP = -1;
  String BestCaseRecord="", BCPC="", BCPN="", BCAtt="", BCNR="";
  int BCR = -1;

  //Constructor
  public ActualAgent(String AN, String PN, String BS, String ATT, String AW, String MAX,
                                                    String MIN, String SDR, Vector LBS){
    //initialize common attributes and behaviour
    super(AN,PN,BS,ATT,AW,MAX,MIN,SDR,LBS);
    this.AW = Double.parseDouble(AW);
    //User's attitude, SDR Max and Min price are saved
    StringTokenizer tokens = new StringTokenizer(ATT,",");
    while(tokens.hasMoreTokens())
    {
       IT = Double.parseDouble(tokens.nextToken());
       IP = Double.parseDouble(tokens.nextToken());
```

```
        IC = Double.parseDouble(tokens.nextToken());
    }
        ASDR = Double.parseDouble(SDR);
        UserMax = Double.parseDouble(MAX);
        UserMin = Double.parseDouble(MIN);

    //Best Relevant case is found for the agent to adjust the Mental Attitude
    BestCaseRecord = C.getBestCase(BS, PN, IT, IP, IC);
    //Needed information are tokenized from best result to find agent's MaxMin
    TokenizeTheBestCase(BestCaseRecord);
    //If there is no match
    if(BestCaseRecord == null){
        System.out.println("no relavent case found... ");
        AdjustedIT = IT;
        AdjustedIP = IP;
    }
    else{
      AdjustAttitude(IT, BCIT, IP, BCIP, BCR);
    }
  }//END OF CONSTRUCTOR ActualAgent

//Needed information are tokenized from best result to find agent's MaxMin
public void  TokenizeTheBestCase(String BestCaseRecord)
{
     StringTokenizer tok = new StringTokenizer(BestCaseRecord,"|");
       while(tok.hasMoreTokens())
       {
         BCPC    = tok.nextToken();
         BCPN    = tok.nextToken();
         BCWarr = Double.parseDouble(tok.nextToken());
         BCPMax = Double.parseDouble(tok.nextToken());
         BCPMin = Double.parseDouble(tok.nextToken());
         BCPrice= Double.parseDouble(tok.nextToken());
         BCAtt   = tok.nextToken();
         BCSDR   = Double.parseDouble(tok.nextToken());
         BCNR    = tok.nextToken();
         BCR = Integer.parseInt(tok.nextToken());
         ThisProductsPPRMax = Double.parseDouble(tok.nextToken());
         ThisProductsPPRMin = Double.parseDouble(tok.nextToken());
       }
      StringTokenizer toke = new StringTokenizer(BCAtt,",");
        while(toke.hasMoreTokens())
        {
          BCIT = Double.parseDouble(toke.nextToken());
```

```
            BCIP = Double.parseDouble(toke.nextToken());
            BCIC = Double.parseDouble(toke.nextToken());
          }
}//End of Method TokenizeTheBestCase

//Adjusting the Agent's (User's) Attitude (Learning...)
public void AdjustAttitude(double IT, double BCIT, double IP, double BCIP, int BCR)
{
     if(BCR == 1) //Best Case negotiation was a successful negotiation
     {
        if ( Math.abs(IT - BCIT)== 0.1)
                AdjustedIT = BCIT;
        else if (BCIT > IT)
                AdjustedIT = IT + 0.1;
        else AdjustedIT = IT - 0.1;

        if( Math.abs(IP - BCIP)== 0.1)
                AdjustedIP = BCIP;
        else if (BCIP > IP)
                AdjustedIP = IP + 0.1;
        else AdjustedIP = IP - 0.1;
     }
     else  //Best Case negotiation was NOT a successful negotiation
     {
        if (BCIT >= IT) AdjustedIT = IT + 0.1;
        else AdjustedIT = IT;
        if(BCIP <= IP) AdjustedIP = IP - 0.1;
        else AdjustedIP = IP;
     }
 }
 public abstract void Notify(Vector listBS, double pprMax, double pprMin);
}// End of abstract class ActualAgent


/*************************************
Abstract class CloneAgent is a subclass of abstract class Agent. Here we have the most
common attributes and methods of BuyerClone & SellerClone. As we need to have Threads for
BuyerClone and SellerClon Agents we implement Runnable here
*************************************/

import java.util.Vector;

public abstract class CloneAgent extends Agent implements Runnable
{
```

```
  //initialize common attributes and behaviour
  public CloneAgent(double aip, double ait, double amaxp, double aminp){
    super(aip,ait,amaxp,aminp);
  }

  public void Notify(double sdr,Vector listBs){
   // do nothing
  }

  public abstract void Offer(double price);
  public abstract void CounterOffer();
}// End of class CloneAgent


/*****************************************
BuyerAgent is a subclass of abstract class ActualAgent. Here we have all the specific
functionalities of a Buyer Agent.  They are Finding Maximum and Minimum price (based on
agents mental attitude, SDR, PPR and Experience)
*****************************************/

import java.util.*;

public class BuyerAgent extends ActualAgent
{
  double AgentMaxPrice = 0.0, AgentMinPrice = 0.0;
  long SLEEP_TIME = 500;
  boolean donedeal = false, isSellerCloneSet = false;
  Vector listOfBuyerClone = new Vector(10);//Keep track of BuyerClones
  Vector listOfNewClone = new Vector(10); //Keep track of New Clones BuyerClone to be
                                          //created, if needed
  Vector priceResult = new Vector(10);    //Negotiation result of each clone in a BuyerAgent
  SellerAgent bestPricedSeller = null;
  PriceResult bestPriceResult;

//Constructor
   public BuyerAgent(String AN, String PN, String BS, String ATT, String AW, String MAX,
                                                    String MIN, String SDR, Vector LBS ){
      //initialize common attributes and behaviour
      super(AN,PN,BS,ATT,AW,MAX,MIN,SDR,LBS);
      this.start();
      FindBuyerMaxMin();
   }

   //This method calcultes the Buyer Agent's Price Ranges
```

```
public void FindBuyerMaxMin()
{
 AgentMaxPrice =  ThisProductsPPRMax - (((ThisProductsPPRMax*AdjustedIP)*(1 - IC))/3) +
            ThisProductsPPRMax*(1 - ASDR)* 05 + ThisProductsPPRMax*(AW-BCWarr)*0.05;
 AgentMinPrice =  ThisProductsPPRMin + (AdjustedIT*(ThisProductsPPRMax -
                                                    ThisProductsPPRMin))/10;
 if(AgentMaxPrice > UserMax)AgentMaxPrice = UserMax;
 System.out.println("Change In Environment: " + AN + " Max Price: " + AgentMaxPrice + " Min
                                                    Price: " + AgentMinPrice);
}

//Here, when Notify is invoked from the EnvironmentWatcher, we create the BuyerClones for
//buyers and ask the SellersAgents to create a corrosponding SellerClones to
//communicate(negotiate) for the product.  As the EnvironmentWatcher class sends all the
//buyers and seller that is found in the market for a product, we want to identify the
//Sellers who are not in the listOfSeller and also the Sellers who are NOT in the
//priceResult Vector. If we find any such new Sellers we create a new BuyerClone and ask the
//New Seller to create a SellerClone for negotiation. Now we keep the reference on both the
//buyerclone and sellerclone.  Then add the new BuyerClone to the istOfBuyerClone and
//listOfNewClone. When StartNegotiation is issued from Environment these new clones will
//start the negotiation

  private void createClone(){
          listOfNewClone.clear();
             boolean isCloneNeeded   = false;
             boolean isNotFoundInPR  = true;
             boolean isNotFoundInBCL = true;

             for(int i = 0 ; i <LBS.size(); i++){
               Object obj = LBS.get(i);
                    if(obj instanceof SellerAgent){
                     if(listOfBuyerClone.size()>0 && priceResult.size()>0){
                    isCloneNeeded = true;
                 }
                           else if(listOfBuyerClone.size() <= 0 && priceResult.size() <=0 ){
                               isCloneNeeded = true;
                           }
                           else if(listOfBuyerClone.size() > 0 && priceResult.size() <=0 ){
                               isCloneNeeded = true;
                           }
                SellerAgent sAgent = (SellerAgent) obj;
                              for(int j = 0; j< listOfBuyerClone.size(); j++){
                               BuyerClone clone = (BuyerClone)listOfBuyerClone.get(j);
```

```
                    if(clone.getSellerName().equalsIgnoreCase(sAgent.getAgentName()) ){
                     isNotFoundInBCL = false;
                    }
                        }
                           for(int k=0; k<priceResult.size(); k++){
                                   PriceResult pr = (PriceResult)priceResult.get(k);
                                   if((pr.AA.AN).equals(sAgent.AN)){
                                  isNotFoundInPR = false;
                               }
                            }
                            if(isNotFoundInBCL && isNotFoundInPR && isCloneNeeded){
                                BuyerClone bClone = new BuyerClone(this,AdjustedIP,
                                                AdjustedIT,AgentMaxPrice,AgentMinPrice);
                                SellerClone sClone = sAgent.setBuyerClone(bClone);
                                bClone.setSellerClone(sClone);
                                listOfBuyerClone.addElement(bClone);
                                listOfNewClone.addElement(bClone);
                            }//if ends
                           isCloneNeeded = false;
                           isNotFoundInPR = true;
                           isNotFoundInBCL = true;
                       }//if is over for listOfBuyerClone is not zero and priceResult not zero
              }
  }// end of method CreateClone


//When a BuyerClone or a SellerClone finishes negotiation with the opponent Clone they
//invoke this method to take the corresponding Seller out of the listOfBuyerClone and to add
//the reference of the Seller, price, deal done or not and negotiation offer/counter-offer
//as PriceResult instance in priceResult Vector.

  public void RemoveNegotiationBuyerClone(PriceResult pResult, BuyerClone bc)
  {
       priceResult.addElement(pResult);
       for(int i=0; i < listOfBuyerClone.size(); i++){
          if(bc.equals((BuyerClone)listOfBuyerClone.get(i))){
                    listOfBuyerClone.remove(i);
              }
       }
         if(listOfBuyerClone.size() == 0){
               for(int j = 0; j<priceResult.size(); j++){
                   PriceResult prpr = (PriceResult)priceResult.get(j);
                   System.out.println(AN + "'s Negotiation recorded with " + prpr.AA.AN + ":
                                                                    " + prpr.negotiationRec);
```

```
                    }
                }
    }

//This method is invoked by BuyerClone(s) RunNegotiation Theard when they are done with
//their deal. Since the RemoveNegotiationBuyerClone method above already removes the Buyer
//clone from listOfBuyerClone when the negotiation clones reach a result (as accepted or
//rejected), at one point listOfBuyerClone will become empty! When listOfBuyerClone becomes
//empty, this method finds the best seller using SuccessfulDealByClone and if exist, sends a
//message to that best Seller saying WantToBuy.
//if the seller replies "DoneDeal"-->All the other sellers who made a successful deal with
//this Buyer's Clones are notified(it is in the priceResult Vector) asking them to remove
//this Buyer from their list.
//if the seller replies "NoDeal"-->We remove the bestSeller from priceResult Vector and go
//for the next bestSeller if there is one exist.
//If the seller replies "Wait"--> We call the wait method, which will only bring in the
//reply of "DoneDeal" or "NoDeal"
//If the BuyerCloneList is empty then we know that we have done with all the clones
//(negotiation).
//Now send a message to Seller if I have an accept in my priceResult Vector

  public void WantToBuy(){

       ActualAgent actuSAgent = null;
       ActualAgent bestPriceSeller = null;
           if(listOfBuyerClone.size() == 0){
           String resultOfSeller = "";
           //If at least one Seller willing to sell
         while(SuccessfulDealByClone() && (!donedeal)){
           resultOfSeller = bestPricedSeller.WantToBuy(this);
           if(resultOfSeller.equals("DealDone")){
           //Notify all the accepted Sellers (by Clones) that you reject them...
           donedeal= true;
           for(int m=0; m<priceResult.size(); m++){
                PriceResult P_R = (PriceResult)priceResult.get(m);
                SellerAgent sssaaa = (SellerAgent)P_R.AA;
                if( (P_R.accRej == 1) && (!((sssaaa.AN).equals(bestPricedSeller.AN))) ){
                     sssaaa.RejectionFromBuyers(this);
                     priceResult.remove(m);
                }
         }
         C.UpdateOrAddNegotiation(SetStringForUpdation());
         System.out.println(AN + " Removes himself from the Market");
         FA.RemoveBuyerSeller(PN, AN, BS);
```

```
      }
      if(resultOfSeller.equals("NoDeal")){
        for(int j=0; j<priceResult.size(); j++){
           PriceResult pprr = (PriceResult)priceResult.get(j);
             if((pprr.AA.AN).equals(bestPricedSeller.AN)){
                 priceResult.remove(j);
              }//if over
        }//for over
      }

      if(resultOfSeller.equals("Wait")){
        resultOfSeller = CallWait(bestPricedSeller);
        if(resultOfSeller.equals("DealDone")){
           for(int m=0; m<priceResult.size(); m++){
              PriceResult P_R = (PriceResult)priceResult.get(m);
              SellerAgent sssaaa = (SellerAgent)P_R.AA;
              if( (P_R.accRej == 1) && (!((sssaaa.AN).equals(bestPricedSeller.AN))) ){
                 sssaaa.RejectionFromBuyers(this);
                 priceResult.remove(m);
              }
           }
          C.UpdateOrAddNegotiation(SetStringForUpdation());
          System.out.println(AN + " Removes himself from the Market");
          donedeal = true;
          FA.RemoveBuyerSeller(PN, AN, BS);
      }
      if(resultOfSeller.equals("NoDeal")){
         for(int j=0; j<priceResult.size(); j++){
             PriceResult pprr = (PriceResult)priceResult.get(j);
             if((pprr.AA.AN).equals(bestPricedSeller.AN)){
               priceResult.remove(j);
              }//if over
         }//for over
     }
   }
}//end while
//if any successful negotiation exist
if(!SuccessfulDealByClone())
 {
   System.out.println("No deal from any Seller for this Buyer: " + AN);
   System.out.println(AN + " Removes himself from the Market");
   FA.RemoveBuyerSeller(PN, AN, BS);
 }
}//end of if
```

```
}//end of method WantToBuy

//This method will return a string of "DoneDeal" or "Nodeal" when a seller agent decides to
//sell or not sell the product to this agent. In the mean time seller will say wait and this
//methods puts this Buyer to sleep for the indicated seconds.

 public String CallWait(SellerAgent bestPricedSeller){
         String resultOfSeller = "Wait";
         while(resultOfSeller.equals("Wait")){
     try{
         this.sleep(3000);
         }catch(InterruptedException ee){
            ee.printStackTrace();
          }
          resultOfSeller = bestPricedSeller.WantToBuy(this);
         }
         return resultOfSeller;
 }//end of method CallWait

//Buyer finds the bestSeller from the priceResult Vector and sets the attribute
//bestPricedSeller with that bestSeller, if not false value is send back to the invoker(Want
//to buy in this case).

public boolean SuccessfulDealByClone(){
   double bestprice = 100000.00;
   PriceResult PR = null;
   bestPricedSeller = null;
   bestPriceResult = null;
       for(int j = 0; j < priceResult.size(); j++){
               PR = (PriceResult)priceResult.get(j);
               if(PR.price < bestprice && PR.accRej == 1){
                  bestprice = PR.price;
                  bestPriceResult = PR;
                  bestPricedSeller = (SellerAgent)PR.AA;
               }
       }//for ends
       if(bestPricedSeller == null)  return false;
       else return true;
 }//end of SuccessfulDealByClone

 //This method sets the string for recording Case Base Reasoning System. It sets the strings
 //in the right order.

 public String SetStringForUpdation(){
```

```
String matcher= "";
        matcher += BS +"|";
        matcher += BCPC +"|";
        matcher += PN +"|";
        matcher += AW +"|";
        matcher += ThisProductsPPRMax +"|";
        matcher += ThisProductsPPRMin +"|";
        matcher += Rounding.toString(bestPriceResult.price,2) +"|";
String att = Rounding.toString(AdjustedIT,2) + "," + Rounding.toString(AdjustedIP,2) +
                                                                                "," + IC;
        matcher += att + "|";
        matcher += Rounding.toString(ASDR,4) +"|";
        matcher += bestPriceResult.negotiationRec +"|";
        matcher += bestPriceResult.accRej;
    System.out.println(matcher);
 return matcher;
}//end of method SetStringForUpdation

//This mehtod is called by StartNegotiation to start the new BuyerClones' negotiation-if
//there was any new Sellers came into the market, which would have been in listOfNewClone
//when createClone was invoked from Notify.

 public void runNow(){
 try{
     Thread.sleep(SLEEP_TIME);
    }catch(InterruptedException ee){
       ee.printStackTrace();
     }
     for(int i = 0; i < listOfNewClone.size(); i++){
             BuyerClone bClone     = (BuyerClone)listOfNewClone.get(i);
             bClone.run();
     }
  }//end of method runNow


//This method is invoked when there is a change in the marketplace. When there is a buyer or
//seller comes in for a product, or removes from a products list in BSL class, and if it
//causes considerable change in Supply Demand ratio for a product here we call FindMaxMin
//method and createClone method. When there is a change only in Public Price Range for a
//product in the marketplace, FindMaxMin method is invoked. Otherwise if its only change in
//the Buyer Seller coming into the market or leaving the market then we just call the Create
//Clone method.

  public void  Notify(double sdr, Vector listBS){
```

```
  BuyerClone allBuyerClone = null;
      LBS = listBS;
      createClone();
      if((Math.abs(ASDR-sdr)>0.1)){
       ASDR = sdr;
       FindBuyerMaxMin();
         for(int i=0; i<listOfBuyerClone.size(); i++){
           allBuyerClone = (BuyerClone)listOfBuyerClone.get(i);
           allBuyerClone.Notify(AgentMaxPrice,AgentMinPrice);
         }
      }
 }// end of method Notify


 public void Notify(Vector listBS, double pprMax, double pprMin){
   BuyerClone allBuyerClone = null;
   if(!(ThisProductsPPRMax == pprMax) || !(ThisProductsPPRMin == pprMin)){
       ThisProductsPPRMax = pprMax;
       ThisProductsPPRMin = pprMin;
       FindBuyerMaxMin();
          for(int i=0; i<listOfBuyerClone.size(); i++){
            allBuyerClone = (BuyerClone)listOfBuyerClone.get(i);
            allBuyerClone.Notify(AgentMaxPrice,AgentMinPrice);
          }
   }
  }// end of another Notify

 //This method is called from EnvironmentWatcher to Start the negotiation between
 //BuyerClone & SellerClone

 public void startNegotiation(){
    runNow();
 }
}// END OF CLASS BuyerAgent


/****************************************
SellerAgent is a subclass of abstract class ActualAgent. Here we have all the specific
functionalities of a Seller Agent.  They are Finding Maximum and Minimum price (based on
agents mental attitude, SDR, PPR and Experience)
****************************************/

import java.util.*;
```

```
public class SellerAgent extends ActualAgent
{
  double AgentMaxPrice = 0.0, AgentMinPrice = 0.0;
  boolean productSold  = false, prodsold = false;

  Vector listOfSellerClone = new Vector(10);//Keep track of SellerClones
  Vector priceResult = new Vector(10);//Negotiation result of each clone in a SellerAgent
  Vector BuyersToBuy = new Vector(10); //Keep track of Buyers who acctually want to buy out
                                       //of the accepted Buyers
  PriceResult pr;
  PriceResult bestPriceResult = null;

  public SellerAgent(String AN, String PN, String BS, String ATT, String AW, String MAX,
                                                       String MIN, String SDR, Vector LBS){
      //initialize common attributes and behaviour
      super(AN,PN,BS,ATT,AW,MAX,MIN,SDR,LBS);
      FindSellerMaxMin();
      this.start();
  }

  //When a SellerClone or a BuyerClone finishes negotiation with the opponent Clone they
  //invoke this method to take the corresponding Buyer out of the listOfSellerClone and to
  //add the reference of the Seller, price, deal done or not and negotiation offer/counter-
  //offer as PriceResult instance in priceResult Vector.

  public synchronized void RemoveNegotiationSellerClone(PriceResult pr, SellerClone sc)
  {
          ActualAgent actuAgent = null;
          ActualAgent bestPriceSeller = null;
          priceResult.addElement(pr);
              for(int i=0; i < listOfSellerClone.size(); i++)
              {
                 if(sc.equals((SellerClone)listOfSellerClone.get(i)))
                           listOfSellerClone.remove(i);
          }
  }// end of method RemoveNegotiationSellerClone

//This method is invoked by BuyerAgent(s) when all their clones are done negotiating with
//SellerClones. Seller adds the Buyers who want to buy the product to a BuyersToBuy list and
//sends the result "Wait" when the listOfBuyers is not empty or when he is waiting for best
//buyer to contact him. But the best buyer of this seller might remove himself by calling
```

```
//the RejectionFromBuyers mehtod from BuyerAgent inside the WantToBuy method of BuyerAgnet.
//This is also taken care of by checking the bestBuyer every time a BuyerAgent calls for
//deal. If the bestbuyer comes in for the product then it will send the message "DoneDeal"
//to the buyer

  public String WantToBuy(BuyerAgent BA){
     boolean addbuyer = true;
     double bestprice = -1.0;
     String resultToBuyer = "";
     BuyerAgent bestBuyer = null;
     bestPriceResult = null;
        if(BuyersToBuy.size()<=0){
                BuyersToBuy.addElement(BA);
        }
        else{
             for(int i = 0; i<BuyersToBuy.size(); i++){
                if(BA.equals((BuyerAgent)BuyersToBuy.get(i))){
                     addbuyer = false;
                }
             }//end of for
             if(addbuyer) BuyersToBuy.addElement(BA);
        }
        for(int k = 0; k<priceResult.size(); k++){
            PriceResult pprr = (PriceResult) priceResult.get(k);
             if(pprr.price > bestprice && pprr.accRej == 1){
                    bestprice = pprr.price;
                    bestPriceResult = pprr;
                         bestBuyer = (BuyerAgent)pprr.AA;
               }

        }
        if(listOfSellerClone.size()<=0){
              if(((bestBuyer.AN).equals(BA.AN)) && (!productSold)){
                  prodsold = true;
                     resultToBuyer = "DealDone";
                     System.out.println(AN + " sold the product to " + BA.AN);
                     for(int k = 0; k<BuyersToBuy.size(); k++){
                      BuyerAgent btb = (BuyerAgent)BuyersToBuy.get(k);
                      if((btb.AN).equals(BA.AN)){
                             BuyersToBuy.remove(k);
                       }
             }
             C.UpdateOrAddNegotiation(SetStringForUpdation());
             FA.RemoveBuyerSeller(PN, AN, BS);
```

```
        }
        else if(!productSold) resultToBuyer = "Wait";
        else if (productSold){
           for(int l = 0; l<BuyersToBuy.size(); l++){
             BuyerAgent bbaa = (BuyerAgent)BuyersToBuy.get(l);
                System.out.println(AN + " rejects " + bbaa.AN);
                   if(bbaa.equals(BA)){
                         BuyersToBuy.remove(l);
                    }
             }
             if(BuyersToBuy.size()<=0){
               System.out.println(AN + " Leave the Marketplace");
               FA.RemoveBuyerSeller(PN, AN, BS);
             }
           resultToBuyer = "NoDeal";
         }
       }//if list of seller clone is over...
       else {
          resultToBuyer = "Wait";
       }
       productSold = prodsold;
       return resultToBuyer;
  }//end of method....

//This method is invoked by BuyerAgent when Buyer finds a deal from some Seller but still
//have some Sellers in his priceResult list as accepted. Buyer invokes this method on such
//other seller(s) to remove himself from their priceResult Vector so that the sellers can
//accept other buyers deal by leaving the "Wait" reply and going into "DoneDeal" or
//"NoDeal".

  public void RejectionFromBuyers(BuyerAgent ba){
       for (int i = 0; i<priceResult.size(); i++){
           PriceResult ppprrr = (PriceResult)priceResult.get(i);
           if((ppprrr.AA).equals(ba)){
              priceResult.remove(i);
           }
       }
  }// end of method RejectionFromBuyers

//This method sets the string for recording Case Base Reasoning System. It sets the strings
//in the right order.

public String SetStringForUpdation(){
    String matcher= "";
```

```
    matcher += BS + "|";
    matcher += BCPC +"|";
    matcher += PN + "|";
    matcher += AW + "|";
    matcher += ThisProductsPPRMax + "|";
    matcher += ThisProductsPPRMin + "|";
    matcher += Rounding.toString(bestPriceResult.price,2) + "|";
    String att = Rounding.toString(AdjustedIT,2) + "," + Rounding.toString(AdjustedIP,2) +
"," + IC;
    matcher += att + "|";
    matcher += Rounding.toString(ASDR,4) + "|";
    matcher += bestPriceResult.negotiationRec + "|";
    matcher += bestPriceResult.accRej;

     System.out.println(matcher);
 return matcher;
}// end of method SetStringForUpdation


//This method is invoked by createClone method of BuyerAgent when this Seller agent is new
//or not in priceResult Vector for that BuyerAgent. It is done this way so that we can have
//the references of each other (Clones) and Clones can have proper references to their
//parent agents (BuyerAgent or SellerAgent).

  public SellerClone setBuyerClone(BuyerClone bClone){
    SellerClone sClone = new SellerClone(this,AdjustedIP,AdjustedIP,AgentMaxPrice,
                                                                    AgentMinPrice);
    sClone.setBuyerClone(bClone);
    listOfSellerClone.addElement(sClone);
   return sClone;
  }// end of method setBuyerClone

 //This method calculates the Seller Agent Price Ranges

  public void FindSellerMaxMin()
  {
    AgentMaxPrice =  ThisProductsPPRMax - (AdjustedIT*(ThisProductsPPRMax -
                                                    ThisProductsPPRMin))/10;
    AgentMinPrice =  ThisProductsPPRMin + ((ThisProductsPPRMin* AdjustedIP)*(1 - IC))/3 +
            ThisProductsPPRMin *(1 - ASDR)*0.05 + ThisProductsPPRMin *(AW-BCWarr)*0.05;
    if(AgentMinPrice > UserMin)AgentMinPrice = UserMin;
      System.out.println("Change In Environment: " + AN + " Max Price: " + AgentMaxPrice + "
                                                Min Price: " + AgentMinPrice);
  }// end of method FindSellerMaxMin
```

```
//This method is invoked when there is a change in the marketplace. When there is a buyer or
//seller comes in for a product, or removes from a products list in BSL class, and if it
//causes considerable change in Supply Demand ratio for a product here we call FindMaxMin
//method. Also, when there is a change only in Public Price Range for a product in the
//marketplace, FindMaxMin method is invoked.

  public void Notify(double sdr,Vector listBs){
    SellerClone allSellerClone = null;
        if((Math.abs(ASDR-sdr)>0.1)){
           ASDR = sdr;
           FindSellerMaxMin();
           for(int i=0; i<listOfSellerClone.size(); i++){
                allSellerClone = (SellerClone)listOfSellerClone.get(i);
                      allSellerClone.Notify(AgentMaxPrice,AgentMinPrice);
           }
        }//end of if
  }//end of Notify

  public void Notify(Vector listBS, double pprMax, double pprMin){
    SellerClone allSellerClone = null;
    if(!(ThisProductsPPRMax == pprMax) || !(ThisProductsPPRMin == pprMin)){
        ThisProductsPPRMax = pprMax;
        ThisProductsPPRMin = pprMin;
        FindSellerMaxMin();
           for(int i=0; i<listOfSellerClone.size(); i++){
             allSellerClone = (SellerClone)listOfSellerClone.get(i);
             allSellerClone.Notify(AgentMaxPrice,AgentMinPrice);
           }
    }
  }
}// End of class SellerAgent


/************************************
BuyerClone is a subclass of abstract class AgentClone. Here we have all the specific
functionalities of a BuyerClone.  They are finding the BidIncrement, Calculating Offer and
Counter Offer, and Recalculation of BidIncrement when changes are notified either in Public
Price Range or Supply Demand Ratio or in new buyer or seller comes in.
************************************/

public class BuyerClone extends CloneAgent
{
  double bidInc = 0.0, lastPrice = 0.0;
```

```
 boolean rejectCondition = false;
 boolean isNotDone = true;
 boolean setOffer = false;
 boolean setCounterOffer = false;
 String negotiationRec ="";
 SellerClone sClone = null;
 BuyerAgent bAgent = null;
 PriceResult priceResult;//To save the negotiation result of this Buyer Clone

 public BuyerClone(BuyerAgent agent , double aip, double ait, double amaxp, double aminp){
       //initialize common attributes and behaviour
       super(aip,ait,amaxp,aminp);
       bidInc = BidIncrement(aip, ait, amaxp, aminp);
       bAgent = agent;
       RunNegotiation runNeg = new RunNegotiation();
       runNeg.start();//Start the negotiation therad
 }

//This method finds the increment price values of Buyer when they are created and also when
//they are notified of changes in the market.

 double BidIncrement(double aip, double ait, double amaxp, double aminp){
      bidInc = (amaxp - aminp)/5 + 0.5*(ait*(amaxp - aminp)/10*aip);
         // System.out.println("Bid increment for buyer: " + bidInc);
      return bidInc;
 }

 public void setSellerClone(SellerClone sClone){
            this.sClone =  sClone;
 }
 public String getSellerName(){
       return sClone.getSellerName();
 }

//This  Started  when  BuyerClone  is  created  But  actual  negotiation  offer  starts  when
//BuyerAgent's runNow() invokes it again

 public void run(){
              negotiationRec += Rounding.toString(aminp,2);
              lastPrice = aminp;
              setOffer = true;
 }
```

```
//Offer increments are done here with saving the offer/counter-offer into the string
//negotiationRec for later use through priceResult instance. In case Accepted or Rejected
//corresponding SellerClone as well as this Clone is notified by calling
//CloneNegotiationAcceptReject method.

  public void Offer(double price)
  {
    if( price <= amaxp){
      negotiationRec += "/" + Rounding.toString(price,2);
         lastPrice = price;
         sClone.CloneNegotiationAcceptReject("accepted");
         CloneNegotiationAcceptReject("accepted");
    }
  else{
       negotiationRec += "/" + Rounding.toString(price,2);
        setCounterOffer = true;
       }
  }

//Offer increments are done here with saving the offer/counter-offer into the string
//negotiationRec for later use through priceResult instance. In case Accepted or Rejected
//corresponding SellerClone as well as this Clone is notified by calling
//CloneNegotiationAcceptReject method.

  public void CounterOffer(){
      aminp = aminp + bidInc;
      if(aminp > amaxp)
      {
          aminp = amaxp;
          rejectCondition =  true;
      }
    try{
        Thread.sleep(3000);
         }catch(InterruptedException ee){
            ee.printStackTrace();
         }
         if(!rejectCondition){
            negotiationRec += "/" + Rounding.toString(aminp,2);
            lastPrice = aminp;
             sClone.Offer(aminp);
         }
         else {
              sClone.CloneNegotiationAcceptReject("rejected");
```

```
            CloneNegotiationAcceptReject("rejected");
        }
    }

//When chnges occur in the environment this method is asked to change the to the new maximum
//and minimum price.

    public void Notify(double amaxp, double aminp){
        this.amaxp = amaxp;
        this.aminp = aminp;
    }

    //This method is used to control the flow of the offer and counter offer method.

    public synchronized void doNegotiation(){
      if(setOffer){
          sClone.Offer(aminp);
          setOffer = false;
      }
        if(setCounterOffer){
            CounterOffer();
            setCounterOffer = false;
        }
    }

//When a negotiation in offer/counter-offer method is over this method is called to save the
//needed result and to create the instance of PriceResult with accepted or rejected price,
//negotiation record string, result of negotiation and reference to the SellerAgent whom
//this Clone was talking to.

    public void CloneNegotiationAcceptReject(String AccRej){

                int result = -1;
                if (AccRej.equals("accepted"))
                    result = 1;
                else result = 0;

                PriceResult pr = new PriceResult();
                pr.price = lastPrice;
                pr.accRej = result;
                pr.AA = sClone.sAgent;
                pr.negotiationRec = negotiationRec;
                bAgent.RemoveNegotiationBuyerClone(pr,this);
                isNotDone = false;
```

```
  }

//This iner class serves as the thread that handles offer and ounter-offer by sleeping (x)
//seconds between offer and counter-offer. It is also responsible for notifying the Buyer
//Agnet to chcek and see if he should start Talking to the Seller using the call
//"bAgent.WantToBuy()".

      class RunNegotiation extends Thread
      {
          RunNegotiation()
          {
              super();
          }
          public void run()
          {
           while(isNotDone)
           {
               try
                   {
                      Thread.sleep(6000);
                   }catch(InterruptedException iEx){iEx.printStackTrace();}

               doNegotiation();
           }
                  bAgent.WantToBuy();
          }
       }
}// End of class BuyerClone


/*****************************************
SellerClone is a subclass of abstract class AgentClone. Here we have all the specific
functionalities of a SellerClone.  They are: Finding the BidDecrement, Calculating Offer and
Counter Offer, and Recalculation of BidDecrement when changes are notified either in Public
Price Range or Supply Demand Ratio or when new buyer or seller comes in.
*****************************************/

public class SellerClone extends CloneAgent
{
    double bidDec = 0.0;
    double lastPrice = 0.0;
    String negotiationRec ="";
    boolean firstOffer = true;
    boolean rejectCondition = false;
```

```
  boolean isNotDone = true;
  boolean setCounterOffer = false;
  SellerAgent sAgent = null;
  BuyerClone bClone = null;
  PriceResult priceResult;//To save the negotiation result of this Clone

 public SellerClone(SellerAgent agent, double aip, double ait, double amaxp, double aminp){
  //initialize common attributes and behaviour
  super(aip,ait,amaxp,aminp);
  bidDec = BidDecrement(aip, ait, amaxp, aminp);
  sAgent = agent;
  RunNegotiation runNeg = new RunNegotiation();
  runNeg.start();//Start the negotiation therad
 }

 public void setBuyerClone(BuyerClone bClone){
           this.bClone = bClone;
 }
 public String getSellerName(){
      return sAgent.getAgentName();
 }

//This method finds the decrement price values of Seller when they are created and also when
//they are notified of changes in the market.

  double BidDecrement(double aip, double ait, double amaxp, double aminp){
       bidDec = (amaxp - aminp)/5 + 0.5*(ait*(amaxp - aminp)/10*aip);
       //System.out.println("Bid decrement for Seller: " + bidDec);
   return bidDec;
  }

//Offer increments are done here with saving the offer/counter-offer into the string
//negotiationRec for later use through priceResult instance. In case Accepted or Rejected
//corresponding BuyweClone as well as this Clone is notified by calling
//CloneNegotiationAcceptReject method.

  public void Offer(double price){
       if( price >= aminp ){
            negotiationRec += Rounding.toString(price,2);
            lastPrice = price;
            CloneNegotiationAcceptReject("accepted");
             bClone.CloneNegotiationAcceptReject("accepted");
       }
       else if(rejectCondition==false){
```

```
                negotiationRec += Rounding.toString(price,2) + "/";
                setCounterOffer = true;
        }
        else if (rejectCondition==true){
             negotiationRec += Rounding.toString(price,2);
             CloneNegotiationAcceptReject("rejected");
             bClone.CloneNegotiationAcceptReject("rejected");
        }
  }

//Offer increments are done here with saving the offer/counter-offer into the string
//negotiationRec for later use through priceResult instance. In case Accepted or Rejected
//corresponding BuyerClone as well as this Clone is notified by calling
//CloneNegotiationAcceptReject method.

  public void CounterOffer(){
    if(!firstOffer){
                amaxp =  amaxp - bidDec;
                if(amaxp < aminp)
                {
                  amaxp = aminp;
                  rejectCondition =  true;
                }
    }
        else{
              firstOffer = false;
        }
        try{
           Thread.sleep(3000);
        }catch(InterruptedException ee){
           ee.printStackTrace();
        }

        negotiationRec += Rounding.toString(amaxp,2) + "/";
        lastPrice = amaxp;
        bClone.Offer(amaxp);
  }

 //When chnges occur in the environment this method is asked to change the to the new maximum
 //and minimum price.

   public void Notify(double amaxp, double aminp){
          this.amaxp = amaxp;
          this.aminp = aminp;
```

```
 }

 //This method is used to control the flow of the offer and counter offer method.

 public synchronized void doNegotiation(){
      if(setCounterOffer){
          CounterOffer();
          setCounterOffer = false;
      }
 }

//When a negotiation in offer/counter-offer method is over this method is called to save the
//needed result and to create the instance of PriceResult with accepted or rejected price,
//negotiation record string, result of negotiation and reference to the BuyerAgent whom this
//Clone was talking to.

    public void CloneNegotiationAcceptReject(String AccRej){
           int result;
              if (AccRej.equals("accepted"))
                  result = 1;
              else
                  result = 0;

              PriceResult pr = new PriceResult();
              pr.price = lastPrice;
              pr.accRej = result;
              pr.AA = bClone.bAgent;
              pr.negotiationRec = negotiationRec;
              sAgent.RemoveNegotiationSellerClone(pr,this);
              isNotDone = true;
 }

//This iner class serves as the thread that handles offer and counter-offer by sleeping (x)
seconds between offer and counter-offer.

class RunNegotiation extends Thread
       {
           RunNegotiation()
           {
               super();
           }
           public void run()
           {
            while(isNotDone)
```

```
            {
                try
                    {
                       Thread.sleep(3000);
                    }catch(InterruptedException iEx){iEx.printStackTrace();}

                doNegotiation();
            }
           }
      }

}// End of class SellerClone
```

Database schema for this application

```
================================

/*==============================================================*/
/* Database name:  PHYSICALDATAMODEL_1                          */
/* DBMS name:      ORACLE Version 7                             */
/* Created on:     8/07/02 11:56:22 AM                          */
/*==============================================================*/

drop table SCOTT.BUYEREXP cascade constraints
/
drop table SCOTT.RANGE cascade constraints
/
drop table SCOTT.SELLEREXP cascade constraints
/
drop table SCOTT.SIMVALUE cascade constraints
/
/*==============================================================*/
/* Table: BUYEREXP                                              */
/*==============================================================*/

create table SCOTT.BUYEREXP (
   PRODCAT              VARCHAR2(15)                  default '1',
   PRODNAME             VARCHAR2(15)                  default '2',
   WARRANTY             FLOAT(15)                     default 3,
   PPRMAX               FLOAT(15)                     default 4,
   PPRMIN               FLOAT(15)                     default 5,
   PRICE                FLOAT(15)                     default 6,
   ATTITUDE             VARCHAR2(15)                  default '7',
```

```
   SDR                  FLOAT(15)                      default 8,
   NEGREC               VARCHAR2(50)                   default '9',
   RESULT               NUMBER                         default 10
)
/

/*==============================================================*/
/* Table: RANGE                                                 */
/*==============================================================*/

create table SCOTT.RANGE (
   PRODUCTCAT           VARCHAR2(15)                   default '1',
   PRODNAME             VARCHAR2(15)                   default '2',
   MINPRICE             FLOAT(15)                      default 3,
   MAXPRICE             FLOAT(15)                      default 4
)
/

/*==============================================================*/
/* Table: SELLEREXP                                             */
/*==============================================================*/


create table SCOTT.SELLEREXP (
   PRODCAT              VARCHAR2(15)                   default '1',
   PRODNAME             VARCHAR2(15)                   default '2',
   WARRANTY             FLOAT(15)                      default 3,
   PPRMAX               FLOAT(15)                      default 4,
   PPRMIN               FLOAT(15)                      default 5,
   PRICE                FLOAT(15)                      default 6,
   ATTITUDE             VARCHAR2(15)                   default '7',
   SDR                  FLOAT(15)                      default 8,
   NEGREC               VARCHAR2(50)                   default '9',
   RESULT               NUMBER                         default 10
)
/


/*==============================================================*/
/* Table: SIMVALUE                                              */
/*==============================================================*/

create table SCOTT.SIMVALUE (
   PRODCAT1             VARCHAR2(15)                   default '1',
```

```
   PRODCAT2              VARCHAR2(15)                default '2',
   VALUE                 FLOAT(15)                   default 3
)
/
```

VITA AUCTORIS

Osmand Christian was born in 1974 in Jaffna, Sri-Lanka. He graduated from St. Brebeuf High School, Toronto, in 1996. From there he went to the University of Windsor where he obtained a B.Sc. in Computer Science in 2000. During the year of 2000 he worked as a System Analyst for AIG inc. in Manhattan, New York. Currently he is a candidate for the Master's degree in Computer Science at the University of Windsor and hopes to graduate in October 2002.