Design and implementation of a multiagent stock trading system

Design and Implementation of a Multi-agent Stock Trading System

Geylani Kardas[1], Moharram Challenger, Suleyman Yildirim, and Ali Yamuc

International Computer Institute, Ege University, 35100, Bornova, Izmir, Turkey
geylani.kardas@ege.edu.tr, moharram.challenger@mail.ege.edu.tr, slmnyildirim@gmail.com, aliyamuc@gmail.com

Abstract

Stock trading is one of the key items in economy and estimating its behavior and taking the best decision in it are among the most challenging issues. In order to cope with those challenges, solutions based on intelligent agent systems are proposed. Agents in a Multi-agent System (MAS) can share a common goal or they can pursue their own interests. That nature of MASs exactly fits to the requirements of free market economy. Although existing studies include noteworthy proposals on agent based market simulation and/or researchers discuss theoretical design issues of agent based stock exchange systems, unfortunately only a very few of the studies considers exact development and implementation of multi-agent stock trading systems within the software engineering perspective and guides to the software engineers for constructing such software systems starting from the scratch. In order to fill this gap, in this paper, we discuss development of a multi-agent based stock trading system by taking into consideration of software design according to a well-defined agent oriented software engineering methodology and implementation with a widely-used MAS software development framework. Each participant in the system is first designed as Belief-Desire-Intention (BDI) agents with their facts, goals and plans and then BDI reasoning and behavioural structure of the designed agents are implemented. Lessons learned during design and development within the software engineering perspective and evaluation of the implemented multi-agent stock exchange system are also reported.

Keywords: Software agents, Agent-oriented software engineering, Stock Trading, Belief-Desire-Intention.

[1] Corresponding author. Tel: +90-232-3423232-3223

1. Introduction

Design and development of software systems based on intelligent agents keep their emphasis on both artificial intelligence and software engineering research areas. In its widely-accepted definition, an agent is an encapsulated computer system (mostly a software system) situated in an environment, and that is capable of flexible autonomous actions in this environment in order to meet its design objectives [1]. These autonomous, responsive and proactive agents have also social abilities and interact with other agents and humans in order to complete their own problem solving. They may also behave in a cooperative manner and collaborate with other agents to solve common problems. To perform their tasks and interact with each other, intelligent agents constitute systems called Multi-Agent Systems (MAS).

On the other hand, stock trading is one of the key items in economy and estimating its behavior and taking the best decision in it are among the most challenging issues. In order to cope with those challenges, various studies exist (e.g. [2], [3], [4], [5], [6]) in artificial intelligence and software engineering research areas which propose agent based solutions. Considering aforementioned properties of autonomous agents, use of MASs both in modeling stock trading markets and exact implementation of software systems in real-world stock / trading applications is worthwhile. Interactions of agents between each other can be either cooperative or selfish [7]. In other words, agents can share a common goal or they can pursue their own interests. That nature of MASs exactly fits to the requirements of free market economy.

Although existing studies include noteworthy proposals on agent based market simulation (e.g. [8], [9]) and/or researchers discuss theoretical design issues of agent based stock exchange systems (such as learning, prediction, planning and behavioral model) (e.g. [2], [5], [10]), unfortunately only a very few of the studies (e.g. [3], [6]) considers exact development and implementation of multi-agent stock trading systems within the software engineering perspective and guides to the software engineers for constructing such software systems starting from the scratch. In order to fill this gap, in this paper, we discuss development of a multi-agent based stock trading system by taking into consideration of software design according to a well-defined agent oriented software engineering methodology and implementation with a widely-used MAS software development framework.

Perhaps one of the most important issues, which should be taken into account during realization of agent based stock trading systems, is appropriate modeling of agent behaviors within a software system. Execution order of agent plans is mostly based on the known facts about the environment and these facts dynamically change during the lifecycle of a software agent. Our study also introduces an approach for both modeling and software implementation of each stock trading agent's facts, goals and plans according to the well-known Belief-Desire-Intention (BDI) architecture [11]. We discuss how facts about the current stock market can be represented as agent *beliefs*, how goal of each stock trader can be modeled as agent *desires* and how *intentions* (in other words agent plans) can be constructed in order to achieve goals of stock traders. Software implementation of these components, which is mostly ignored in the related work, is also discussed in this paper. Design and implementation of our proposed system are based on the exact trading rules and procedures of Istanbul Stock Exchange (ISE) [12] which is one of the most important exchanges in the world.

Rest of the paper is organized as follows: Section 2 includes a brief discussion of the stock exchange system that is used as a model for our implemented MAS. Section 3 discusses the software development methodology that we apply during analysis and design of the system and software frameworks and tools employed during the implementation of the system. Development of the stock trading system is discussed in Section 4. Experiences gained during the development are discussed in Section 5. Related work is given in Section 6. Section 7 concludes the paper.

2. Istanbul Stock Exchange (ISE)

In terms of economy, a market is a place that buyers and sellers meet and trade. There are many types of financial markets like capital markets, commodity markets, money markets, insurance markets and foreign exchange markets. Capital markets consists of primary markets which newly formed securities are bought and sold and secondary markets which existing securities may be bought or sold.

In this study, we focus on stock markets which are one of the subtypes of capital markets. A stock market is a market in which long or short term orders and demands are met and executed. A stock exchange provides trading facilities for stock brokers and traders to trade

stock and other securities. Trading rules and procedures of Istanbul Stock Exchange (ISE) [12] are chosen for modeling and simulation of our implemented MAS.

There are three participants in an auction. A seller owns the resources and wishes to obtain as much money as possible. An auctioneer acts as the agent for the seller. A buyer frequently knows more than the seller about the value of the resources and wants to pay as little as necessary [13]. Various protocols exist for auction. William Vickrey, winner of the 1996 Nobel Prize in Economic Sciences, establishes the basic taxonomy of auctions based on the order in which prices are quoted and the manner in which bids are tendered: English or ascending-price auction, Dutch or descending-price auction, first-price sealed-bid auction, and Vickrey or uniform second-price sealed-bid auction.

In addition to abovementioned one-sided auction types, there is double sided auction in which both buyer and seller suggest prices [13]. A continuous double auction is one variant in which many individual transactions are carried on at any single moment and trading does not stop as each auction is finished.

Stock Exchange is an example of the continuous double auction and orders in ISE are executed by two procedures; “Opening session-Single Price” and “Multiple Price-Continuous Auction”. In opening session-single price method, buying and selling orders are gathered and awaited in the system without matching in a predetermined time interval. After time expires, buying and selling orders are matched according to the price level that enables the maximum transaction. The difference between opening session-single price and multiple price-continuous auction is that although orders are able to be matched by the stock exchange system, transactions do not occur until transaction time in opening session-single price. Buying orders which are greater than or equal to the “price level” and selling orders which are less than or equal to “price level” are executed only with this price. Single price method may be executed before, during or after multiple price-continuous auction method. ISE applies opening session in the first 20 minutes of each session [14].

In our study, we implement Multiple Price-Continuous Auction since it is the main procedure of stock exchange in ISE [15]. In this procedure, first an order (e.g. buying order) is accepted by the system. Second, corresponding orders (e.g. selling orders) are checked whether the bid price is matched with offer prices. If it matches, transaction between buyer and seller occurs.

Before discussing multiple prices-continuous auction, it is necessary to explain priority rules that are applied by ISE.

In order to match an offer (sell) with a bid (buy), the system applies some rules. A broker, who wants to sell or buy a stock for a customer, has to add all information that an order must have. Orders are executed by the system considering priority rules which are price and time Priority.

- Price Priority: At first, the system searches for price. Lower price orders have a priority to higher price orders in terms of selling, whereas higher price orders have a priority to lower in terms of buying.
- Time Priority: If buying and selling prices are equal, the order which is added to the system before is executed.

To understand the price and time priority, assume that buy and sell orders are entered to the system as follows (Table 1):

Table 1: Orders of a particular stock entered to the stock exchange system in continuous auction session (from [15]).

Buying side					**Selling side**				
Type	**Order Number**	**Quantity (Lot)**	**Price**	**Time**	**Type**	**Order Number**	**Quantity**	**Price**	**Time**
Buy	1	100	2,23	10:00:00	Sell	1	20	2,26	10:00:00
	2	15	2,23	10:00:01		2	70	2,27	10:00:03
	3	200	2,22	10:00:02		3	80	2,27	10:00:04
	4	40	2,24	10:00:03		4	150	2,25	10:00:04
	5	50	2,21	10:00:04		-	-	-	-

In Table 1, the left side represents “buying orders” along with the information of order number, quantity (lot), price, and time when buying order is added to the system. Similarly, the right side of the table represents “selling orders”. Orders in Table 1 are reorganized by stock exchange system according to priority rules mentioned before.

Table 2 shows contents of a “passive file” of a particular stock. A passive file keeps static stock information provided by buyers and sellers. This information will be used by the agents implemented in this study to make decision and do the transactions.

Table 2: Passive file based on "order" (from [15]).

Buying side		Selling side	
Buying Quantity (Lot)	**Buying Price**	**Selling Price**	**Selling Quantity (Lot)**
40	2,24	2,25	150
100	2,23	2,26	20
15	2,23	2,27	70
200	2,22	2,27	80
50	2,21		

Transition from Table 1 to Table 2 is as follows: Considering the buyer side, Stock Exchange System realizes that the best buying order is the order with price level 2,24 and amount of 40 lots. Notice that given price levels are in Turkish Liras (TL) currency. If we look at the entry time of the order, we can see that it is the 4th entry among other buying orders. Although it is the 4th order, it is placed at the top of the buying side of Table 2 as it is the highest price in terms of buying. There are two stocks with the same price level 2,23 but with different entry time. Because 1st order with the amount of 100 lots is entered to the system before the 2nd order with the amount of 30, 1st order has a Time Priority to the 2nd order. Thus, 1st order is placed at the second row and 2nd order is place at the third row of the buying side of Table 2. Similarly, remaining buying orders with price level 2,22 and 2,21 are placed at the 4th and 5th row of the row of the buying side of Table 2.

When we consider the selling side, the first priority is given to the selling order with the lowest price level 2,25 and amount of 150 lots. According to the lowest price rule in terms of selling, it is placed at the top of the right side of Table 2, which is the selling side. The order with the second highest price level 2,26 are put at the second row according to the price priority. There are two selling orders with the same price level 2,27 but with different entry time. The order with an amount of 70 lots is placed at the second row because of the Time Priority, while the other order with an amount of 80 lots are placed at third row of the Table 2. In this example, there is no matching between the highest buying order and the lowest selling order.

Referring back to the Multiple Price-Continuous Auction, the name "continuous" and "multi-price" come from the fact that *during* this session, buying and selling orders are matched in *different price levels.* As soon as orders are entered to the system, orders are immediately checked by the system. If the order at one side (e.g. buying side) is matched with the orders

waiting at the other side (e.g. selling side), transactions are automatically executed. In order to understand this session, consider again Table 2. Assume that a *selling order* with price 2,24 and 40 lots is entered to the system by a broker. System will search for buying side whether there is a buying order. Selling order with 20 lots and buying order with 40 lots are matched at 2,24 price level. Thus, 20 lots transaction will be executed and remaining 20 lots will be written to the passive file. The result is given in Table 3.

Table 3: Passive file after selling order (Price level: 2,24 TL; Amount: 40 lots) (from [15]).

Buying side		**Selling side**	
Buying Quantity (Lot)	**Buying Price**	**Selling Price**	**Selling Quantity (Lot)**
20	2,24	2,25	150
100	2,23	2,26	20
15	2,23	2,27	70
200	2,22	2,27	80
50	2,21		

Let's continue to our example. If a new *buying order* with price level 2,26 and amount of 200 lots, system will search for selling side whether there is a suitable order and detect that there exits even a better (lower) selling order (2,25). 150 lots transaction will be executed with the price level 2,25. Then 20 lots transaction with price level 2,26 will also be executed, because 200 lots of buying order has not been reached yet. Note that the last transaction's price level is the same level with buying order. Transaction will automatically finish if price level of buying order is less than price level of selling order. Eventually, after buying order entered to the system 150 lots (with price level 2,25) and 20 lots (with price level 2,26) transactions are executed. It is clear that total amount of transaction is 170 lots out of 200 lots. Remaining amount 30 lots with the price level 2,26 is written to the buying side of passive file. Table 4 lists the orders waiting in passive file of a particular stock after the execution.

Eventually, the best price level has changed. Buying order with 30 lots is placed at the top of buying side because it is the best (the highest) price level among buying orders. As for selling side; 170 lots are already executed with the price levels 2,25 and 2,26. Among the remaining orders (price level 2,27), the order with the amount of 70 lots is at top because of time priority. Hence it is placed at the top of the selling side of passive table.

Table 4: Passive file after buying order (Price level: 2,26 TL; Amount: 200 lots) (from [15]).

Buying side		Selling side	
Buying Quantity (Lot)	**Buying Price**	**Selling Price**	**Selling Quantity (Lot)**
30	2,26	2,27	70
20	2,24	2,27	80
100	2,23		
15	2,23		
200	2,22		
50	2,21		

It is also possible to improve or worsen the order. Order improvement means that an investor increases the *buying price* of a particular stock which resides in buying side of the passive file. Similarly, an investor decreases the *selling price* of the stock that resides in buying side of the passive file. As for order worsening, it means that an investor increases the *selling price* of a particular stock which is in the selling side of the passive file. Similarly, an investor decreases the buying price of the stock that is in the buying side of the passive table.

To give an example, consider selling order with price level 2,27 and quantity 70 lots in Table 4. When an investor instructs a broker to *order improvement*, price is decreased. For instance, a new price after order improvement becomes 2,26. It means that an investor is consent to sell the stock lower price. In that case, selling and buying orders are matched at 2,26 price level and 30 lots are sold. The word "improvement" comes from the fact that order improvement helps transactions to be made in lower price levels, whereas it is not good for sellers.

When we consider the order worsening, it is advantageous for sellers because an investor sells the stock with a higher price. However this obstructs the matching and therefore transaction. For example, if an order worsening was instructed to broker, new price level of selling order would be higher than 2,27, e.g. new price equals to 2,28. Thus, no matching would occur between buying order with 2,26 price level and 30 lots.

3. Applied Methodology and Used Technologies during System Development

In order to determine system requirements and realize design of the multi-agent stock trading system, we apply Tropos methodology. Tropos [16] is a well-known agent-oriented software engineering (AOSE) methodology that covers analysis and design of MAS development. Two main properties of Tropos cause us to prefer this methodology: First, in Tropos, the notion of

agent and all related mental notions (for instance goals and plans) are used in all phases of software development, from early analysis down to the actual implementation. Second, Tropos covers also the very early phases of requirements analysis, thus allowing for a deeper understanding of the environment where the software must operate, and of the kind of interactions that should occur between software and human agents [17].

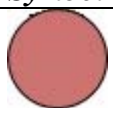
Tropos consists of four phases: *early requirements phase* which provides understanding of overall system, *late requirements phase* in which system-to-be is described, *architectural design phase* which defines the system's global architecture and finally *detailed design phase* which details architectural components of the previous design phase.

Tropos adopts i* [18] modeling framework for analyzing requirements. In i* (which stands for "distributed intentionality"), stakeholders are represented as (social) actors who depend on each other. *Actor diagrams* in Tropos reflect the strategic dependency model for a system. In these diagrams, a dependency represents an "agreement" (called dependum) between two actors—the depender and the dependee. The depender depends on the dependee to deliver the dependum. As stated in [19], the dependum can be a goal (sometimes it is referred as a hard goal) to be fulfilled, a task to be performed, or a resource to be delivered. In addition, the depender may depend on the dependee for a softgoal to be fulfilled. Softgoals represent non-functional system aspects with no clear-cut criteria for their fulfilment (e.g., some of the softgoals for the stock exchange system may be "security of transactions" and "fast operation"). Graphically, actors are represented as circles; dependums —goals, softgoals, tasks and resources— are respectively represented as ovals, clouds, hexagons, and rectangles; and dependencies have the form depender → dependum → dependee.

Actor diagrams are extended during requirements analysis by incrementally adding more specific actor dependencies, discovered by a *means-ends analysis* of each goal [19]. This analysis is specified using another type of diagrams called *rationale diagrams*. A rationale diagram appears as a balloon within which goals of a specific actor are analyzed and dependencies with other actors are established. Goals are decomposed into subgoals and positive/negative contributions of subgoals to goals are specified. Interested readers may refer to [16] and [19] for more information on Tropos methodology.

Table 5 provides a legend for tracing outputs of the system development process which will be introduced later in this paper. Symbols used in our system's actor and rationale diagrams are listed in the table.

Table 5: Symbols used in actor and rationale diagrams throughout the paper

Symbol	***Meaning***
	Actor in a MAS
	A hard goal to be achieved
	A soft goal to be achieved
	A task to achieve a goal
	A resource to be used
	Means-end connection
	Logical AND Decomposition
	Logical OR Decomposition

On the other hand, our stock trader agents are designed according to the Belief-Desire-Intention (BDI) architecture. BDI was first proposed by Bratman [11] and then extended by Rao and Georgeff [20]. In a BDI architecture, an agent decides on which goals to achieve and how to achieve them. *Beliefs* represent the information an agent has about its surroundings, while *Desires* correspond to the things that an agent would like to be achieved. *Intentions*, which are deliberative attitudes of agents, include the agent planning mechanism in order to achieve goals.

In order to implement our BDI agents, JADEX platform is used. JADEX [21] has an agent-oriented reasoning engine for writing rational agents with Extensible Markup Language (XML) and the Java programming language. The development of JADEX agents is based on a

hybrid approach in which declaration of static agent properties and programming of executable agent plans take place. Declaration of static agent properties is given in files called Agent Definition File (ADF). An ADF is written using XML and specifies the BDI model of the related agent. Moreover, agent plans are executable components and they are given in Java program files. The JADEX reasoning engine starts the deliberation process by considering the goals requested by the agent. To this end, it adopts the goals stored in the database that contains all adopted goals by the agent, called the agent's goal-base.

A JADEX goal may be in three states within its lifecycle: New, Adopted or Finished. When an agent is born, creation condition triggers to create initial goal or goals which are in the new state and these are assigned as a main (top-level) goal. Note that a goal in the new state has nothing to do with the agent's deliberation mechanism; it only exits as a candidate goal. An agent must adopt that goal in order to achieve its objectives.

Adopted State consists of three sub states which are *option*, *active* and *suspended*. Once a goal is adopted, it is regarded as a desire and now becomes an *option.* There are three possibilities: an option may be the main (top-level) goal which is added to goal-base, or may be a subgoal of a plan's root goal. If a plan is finished or aborted, unfinished goals are also aborted. Lastly, an option may be a top-level goal created by a plan. In that case, the goal(s) do not rely on the plan that created them. Goal deliberation enables state transitions among adopted goals. A suitable goal is selected as an *active* when this goal is running for particular plan. An active goal maybe *suspended* according to the context condition. If the context of a goal is invalid, that goal will be in suspended state until its context becomes valid. Suspended goal is different from an option goal since it cannot be activated by deliberation mechanism. Whenever the context is valid, suspended goal becomes an option. Thus, it can be pursued by deliberation mechanism [22]. Finally, a goal is in the finished state, when an agent achieves its objective via that particular goal or decides not to perform the execution of that goal. Note that an agent has also the opportunity of dropping a goal and can terminate it.

It is also worth discussing that JADEX has four types of goals: *Perform goal* is related with the actions. Whenever an action is executed or to be done, perform goal is considered to be reached. *Achieve goal* represents the target state that an agent wants to be without specifying how to reach that goal. Agent may use different kinds of plans to accomplish achieve goal. *Query goal* resembles achieve goal to some extent. However it is related to agent's internal

state. An agent may want to retrieve some information that it wants to know. *Maintain goal* is like an update mechanism. The aim of this type of goal is to perceive the environment and the desired state of an agent and re-establish the current state whenever it is violated [23].

Although application of Tropos methodology enhances our system analysis and design steps, a tool support is inevitably needed both for documenting artifacts and using these artifacts actively in the implementation. For this reason, we use a software toolkit called TAOM4E. TAOM4E (Tool for Agent Oriented Modeling) [24] results from re-engineering of the previous version of the TAOM modeler and extends its functionalities, to support the Tropos methodology from early requirements engineering to code generation. The tool supports a model-driven and agent oriented software development. It has been designed to take Object Management Group's (OMG) Model Driven Architecture (MDA) recommendations [25] into account. Tropos system models are developed graphically in TAOM4E's graphical user interface (GUI) and given as an input to the integrated model transformation engine of TAOM4E. Model transformation engine applies the transformation rules on the input model and successfully generates the output model(s) [26]. These capabilities cause us to prefer TAOM4E. Besides, TAOM4E is developed as a plug-in for the widely-used Eclipse platform [27].

4. Development of the Multi-agent Stock Trading System

This section discusses the whole development process of our proposed MAS covering identification of the system needs, design of the system within the AOSE perspective and finally implementation of the system as real BDI agents.

In our case study, individuals (sellers or buyers), send their requests to the stock exchange system[2]. The system processes the requests and returns final results to individuals. This system can be visualized in a Unified Modeling Language (UML) Use Case diagram as shown in Figure 1.

Requests of investors are stored in a table which has two parts, sellers' info and buyers' info. The bid-price for buyers is ordered decreasingly and offer-prices of sellers are stored

[2] stock exchange and stock trading terms are used interchangeably throughout the paper.

increasingly, as described in Tables 1,2,3 and 4 of Section 2. System matches bids and offers according to the rules special for stock exchange as again discussed in Section 2.

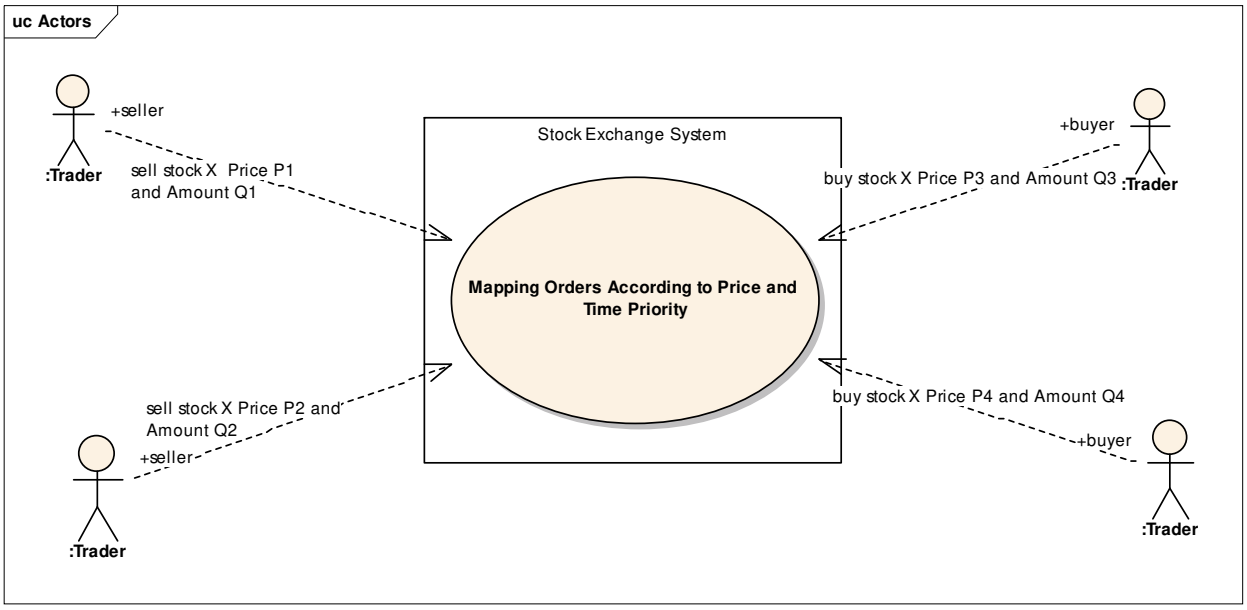


Figure 1: UML Use Case diagram describing the Stock Exchange System from the actors' perspective

In a stock exchange system, investors decide to make a bid/ask order according to the current information coming from another participant called Stock Information System (SIS). These orders are delivered to a broker. The broker starts a deal and negotiates with other brokers via trading floor. Trading floor performs transaction in which bids and asks are matched according to specified rules [15] we previously discuss. In case of a successful transaction, the broker returns a report to the investor. SIS periodically receives system information to be used via investor. Some events like a war in somewhere and indicators like gold or petrol price naturally affect the stock. Hence these events and indicators should be manipulated in the system. Investor agents can make decision based on the known facts by conforming to the BDI architecture. Execution of this system is illustrated in Figure 2.

Development phases of the proposed multi-agent stock exchange system are discussed in the following subsections.

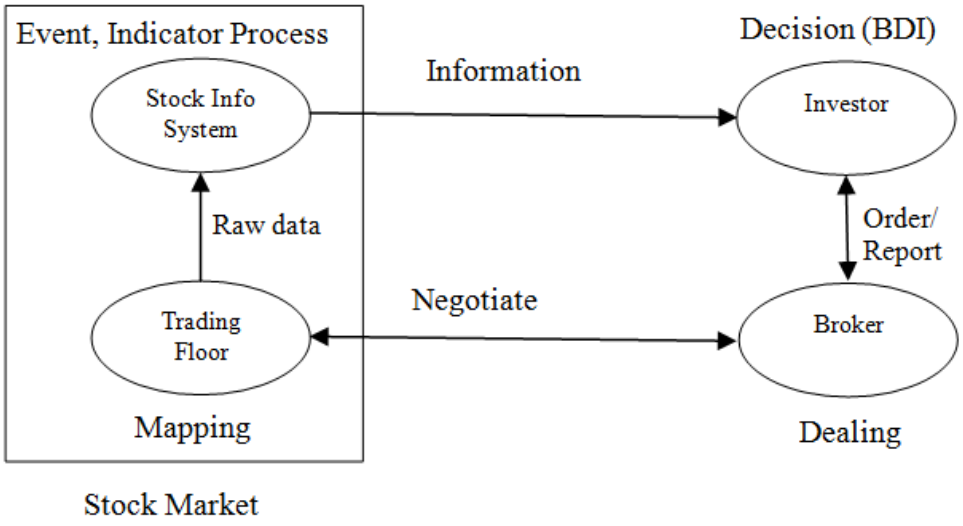


Figure 2: Execution of Stock Exchange System

4.1. Early requirements phase

Concerned with the understanding of a problem by studying an organizational setting; the output of this phase is an organizational model which includes relevant actors, their respective goals and their inter-dependencies. Early requirements include two main diagrams: the actor diagram and the goal diagram. The latter is a refinement of the former with emphasis on the goals of a single actor. Early requirements for our system are shown in Figures 3 and 4, and described as follows:

1. Investor decides to buy or sell a stock according to quantity, price, time and negotiation tolerance, its current strategy and information from SIS. Then it gives the order to the Broker and makes reasoning to decide (see Figure 3).
2. Investor keeps its current strategy or changes it according to the information from SIS (see Figure 4).
3. Investor sends bid/ask order including name of the stock, quantity, price range, and time to the Broker, in order to apply decision that the Broker made (see Figure 3).
4. Broker makes bid(s)/ask(s) to Trading Floor according to parameters of incoming order from Investor and returns a positive or negative acknowledgment to Investor (see Figure 3).
5. Trading Floor places all orders to a table according to price and time priority rules. Then it maps the bid and asks offers, performs transactions and updates the table. Finally, Trading Floor waits for new bids/asks (see Figure 4).
6. SIS fetches economical and political news, statistical reports, social events, and so on from (mostly) Internet or other media and then places them in a database that we call

"knowledge of the world". It also receives instant stock data from Trading Floor. All these raw data are processed with the help of data mining techniques so that this would result in meaningful statistical data to be used by the Investor (see Figure 4).

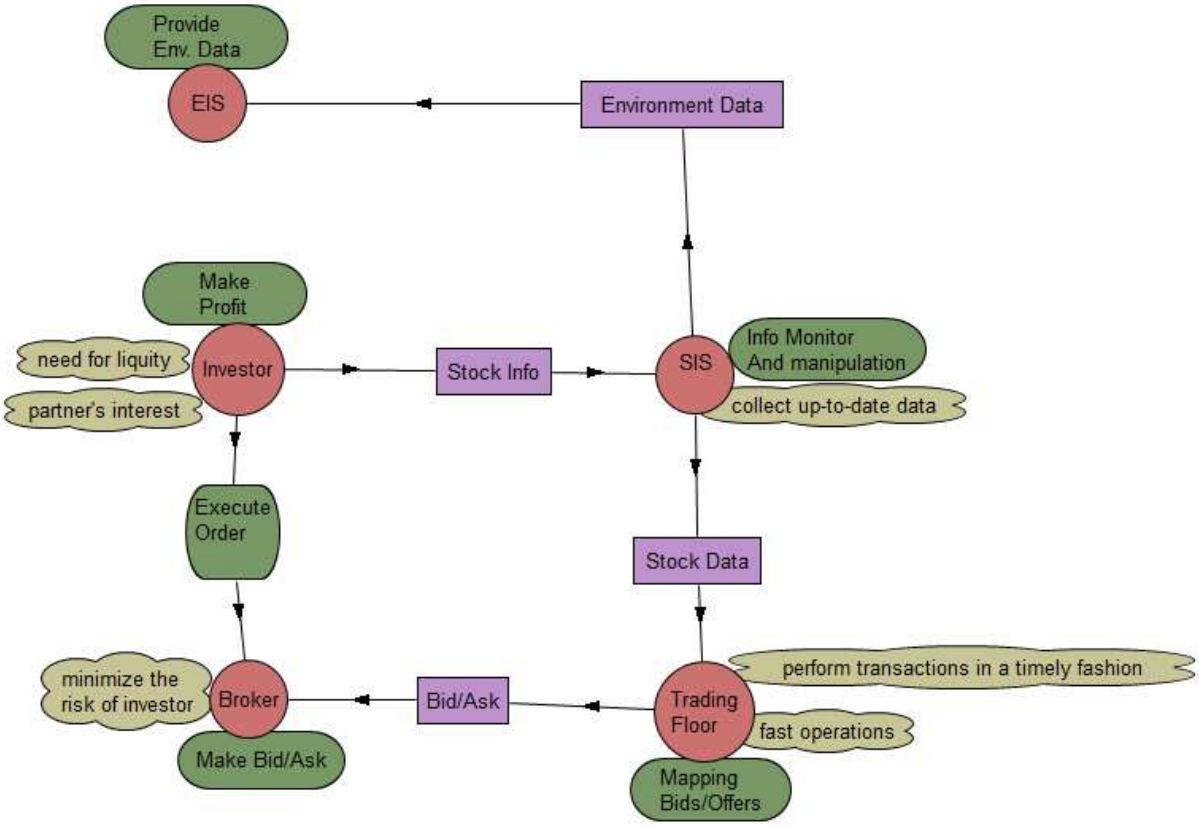


Figure 3: Early Requirements (step 1)

The refined model in Figure 4 elaborates the model in Figure 3 by goal and task decompositions for each actor. For example, the Investor in Figure 4 has a main (hard) goal named "Make Profit" and to achieve this, it needs to use two subgoals namely "Determine Strategy" and "Make Investment Decision". Note that motivation of investors is not limited to "Make Profit". There are other motivations that trigger an investor to buy or sell a stock. We assigned two softgoals to represent these motivations: "*need for liquity*" and "*partner's interest*". Both of these softgoals positively contributes to "Get Stock Info" plan. An investor sells a stock when he/she needs cash (needs a liquity). Similarly, an investor may want to be involved in a company's partnership to earn prestige. Thus we named this softgoal as "partner's interest". Both of "Determine Strategy" and "Make Investment Decision" subgoals need "Get Stock Info" plan as a means. However, "Make Investment Decision" subgoal requires still another plan named "Determine Investment Condition" plan. This plan by itself

is composed of four other subplans: Determine Stock Type, Determine Price Scope, Determine Time Interval and Determine Lot Amount.

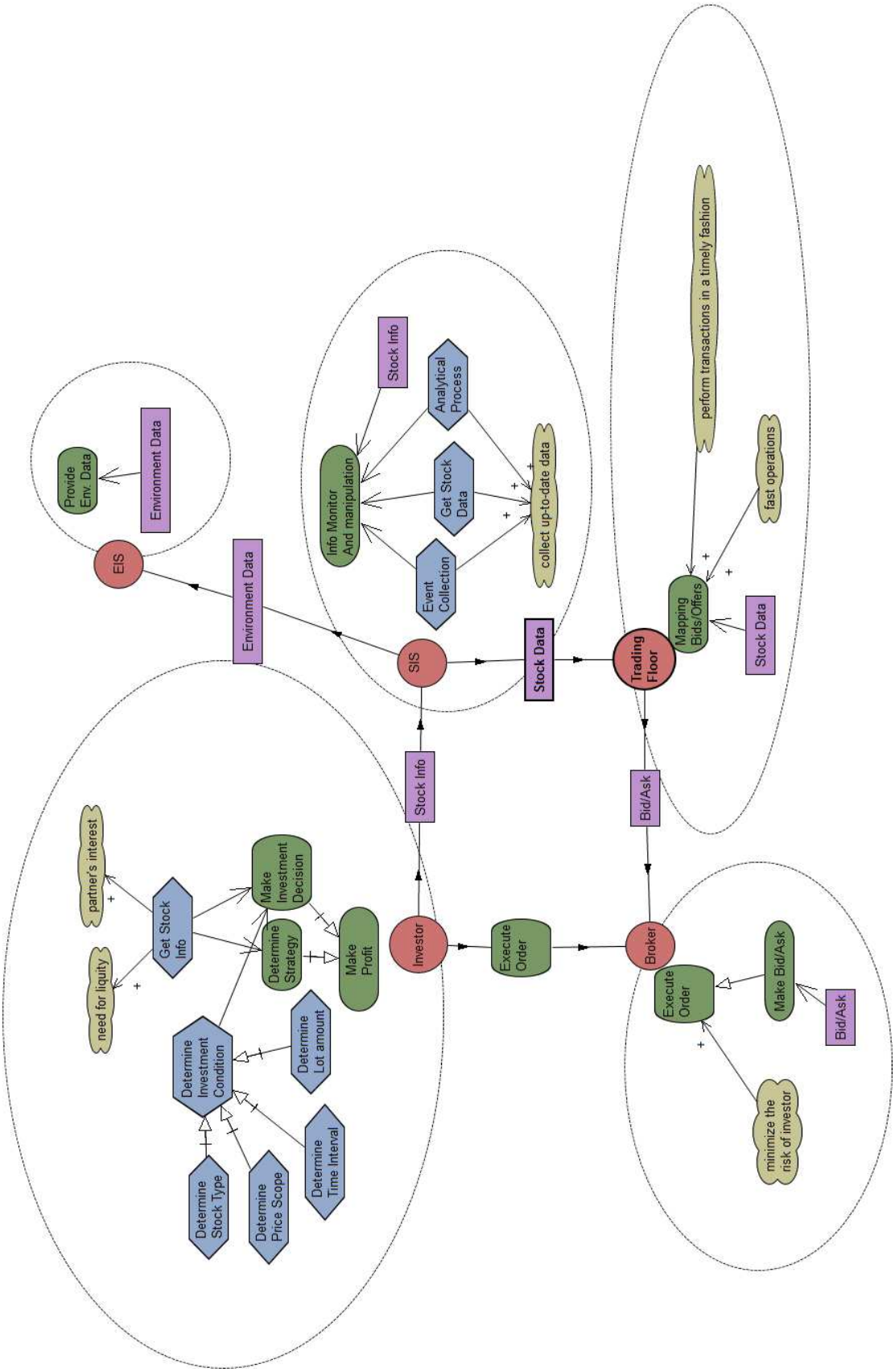


Figure 4: Early Requirements (step 2)

Except for Investor, other actors have also some non-functional aspects. For instance, although Broker's top level goal is to execute an order, he/she has to "*minimize the risk of the investor*" that contributes its execute order root goal. It may take an initiative and suggest some opportunities to the investor to make a good decision. As for SIS actor, it is vital to "*collect up-to-date data*" to which "Event Collection", "Get Stock Data" and "Analytical Process" plans contribute. Lastly, Trading Floor actor have two softgoals related to timing aspects of transactions. In stock trading, one cannot deny the fact that even milliseconds are of extreme importance. "*perform transactions in a timely fashion*" and "*fast operations*" softgoals positively contribute to Mapping Bids/Offers hard goal.

4.2. Late requirements phase

Conforming to the content suggested by Tropos's late requirements phase [16], we add the system actor and analyze its dependencies with other predefined actors in its environment. That enables us to specify system's functional and non-functional requirements as discussed in Section 3.

The system actor in our study is Stock Exchange System (SES). It has dependencies with the actors Investor, EIS, SIS, Trading Floor and Broker. For instance, when we consider the dependencies between SES and Broker, "Bid/Ask Report" is a resource that SES depends on Broker. Broker depends on SES to enable its "Send Bid/Ask" hard goal. SES depends on Broker to its "Do Order" hard goal.

Late requirements for our system are shown in Figures 5 and 6. In the first step of late requirements (Figure 5), the SES is considered with other actors, but in the second step of late requirements (Figure 6), the focus is on the goals of SES. For SES, we introduce the decomposition of the goals, means-ends and we extend the system goals (if needed). For example, when SES has to execute "Get Financial Analysis" hard goal, it needs to execute a plan called "Analysis Process" via a means-ends connection. The plan is decomposed of a number of subplans: "Get Analysis Info", "Process Info" and "Deliver Report". In other words, the mean in this case is the "Analysis Process" which is decomposed into a number of means such as "Get Analysis Info", "Process Info" and "Deliver Report".

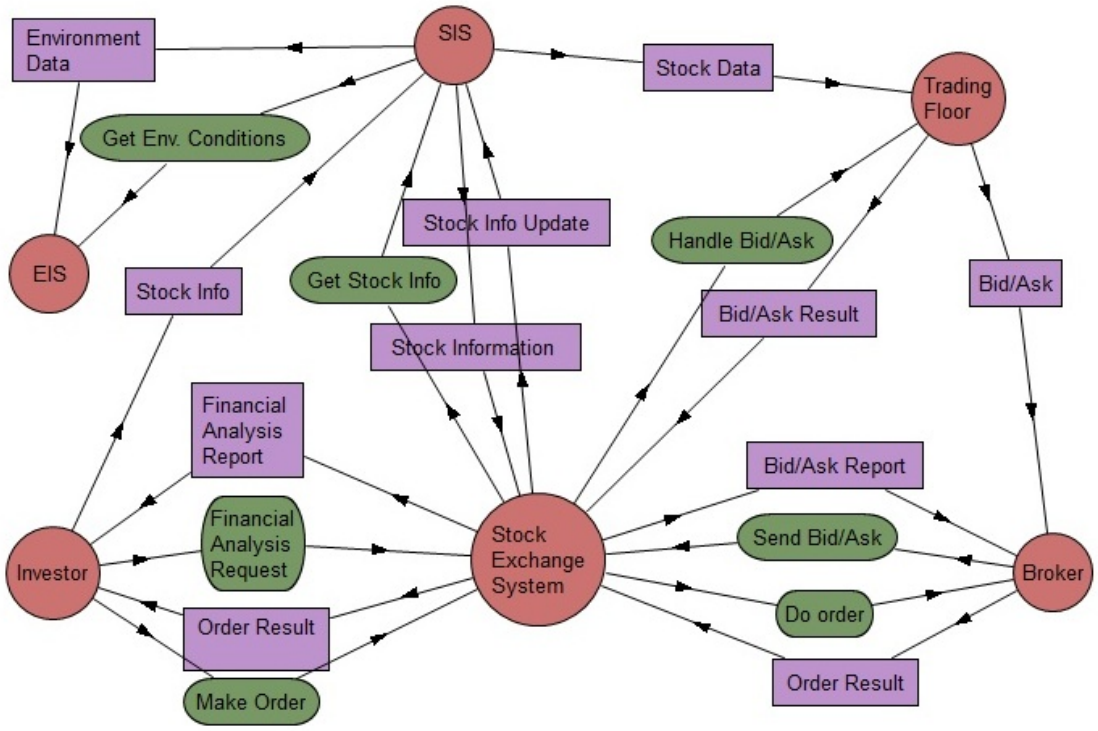


Figure 5: Late Requirements (step 1)

Improved version of the previous model causes the elaboration of goals. For instance, SES has 3 other main hard goals called “Update Stock Info”, “Execute Order” and “Send Bid/Ask” (Figure 6). In fact, detailed analysis of system leads to find more exact comprising components of goals which form the hard goals needed in late requirement phase. The hard goals by themselves need to be decomposed in more detailed parts as deferent tasks which should be performed. Therefore all of the hard goals use several plans as their means to achieve this decomposition.

As well as hard goals, SES has some softgoals as shown in Figure 6. We identified four types of softgoals which emphasize the most important aspects of stock trading. The first one is related with browsing up-to-date data. For updating stock information goal, collecting stock information plan positively contributes to “*search for secure and real-time data”* softgoal. The second one is about reporting and information sharing among agents, as they are critical to keep track of transactions. For that reason, “*fast delivery*” softgoal is delegated to “Deliver Report”, “Deliver Stock Info” and “Deliver Bid/Ask” plans of three different root goals of the system-to-be. The third one is about information retrieval. To execute an order, the system needs to gather the type, amount, price scope and time interval of a particular stock. In order to get these information, SES has to retrieve secure data in a timely fashion as mentioned in

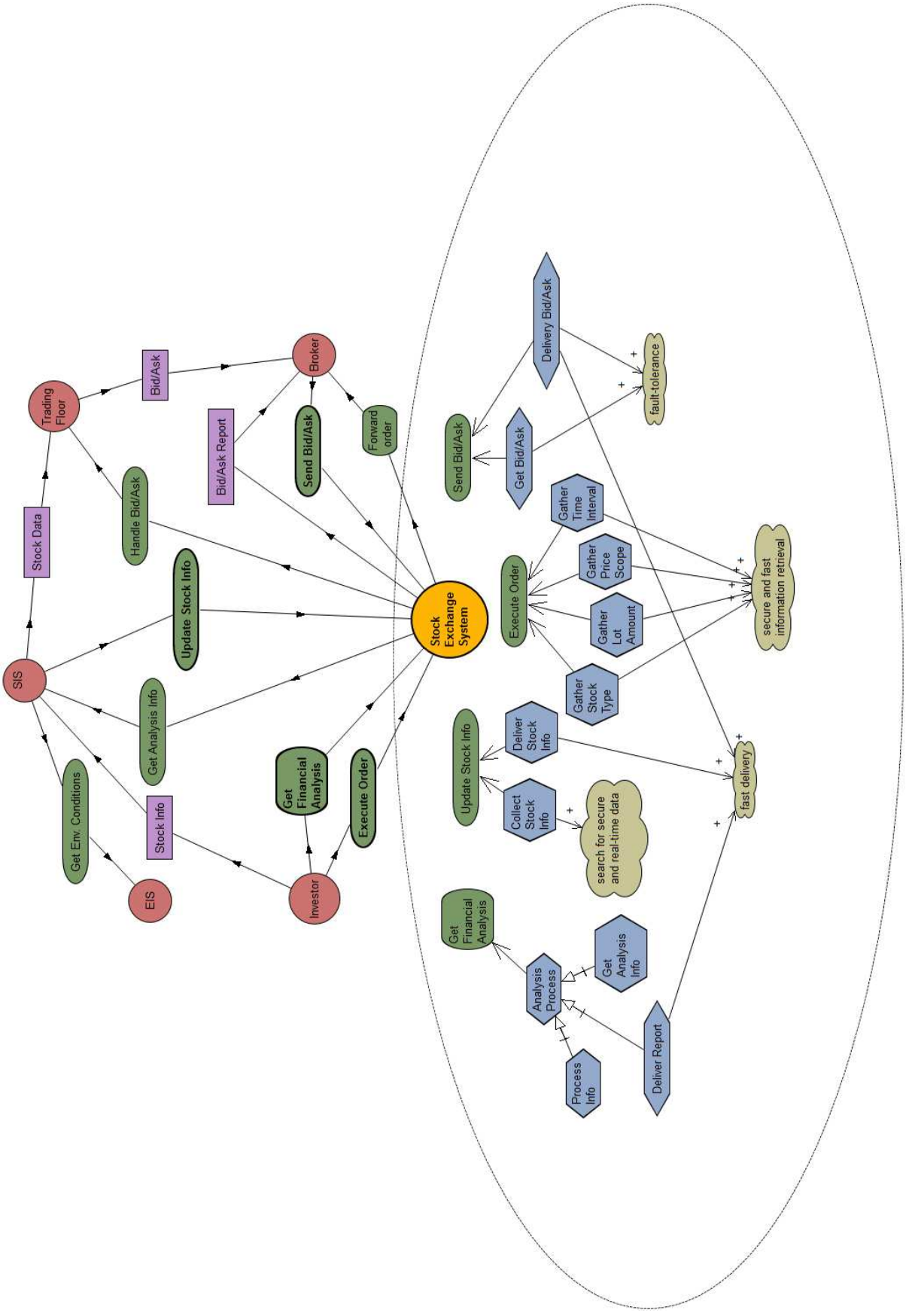


Figure 6: Late Requirements (step 2)

early requirements section. Thus, "*secure and fast information retrieval*" is delegated to four plans of "Execute Order" hard goal. The last softgoal is about exception handling. After broker sends bids or asks to the SES, some errors may occur due to several reasons. In order to cope with this situation, "*fault tolerance*" softgoal is delegated to "Get Bid/Ask" and "Delivery Bid/Ask" plan of "Send Bid/Ask" goal.

4.3. Architectural design phase

The system's global architecture is defined in terms of subsystems interconnected through data, control and other dependencies. Architectural design consists of three steps: 1) decomposing and refining the system actor diagram, 2) identifying actor's capabilities that the actors require to fulfill their goals and plans, 3) creating agents from actors and assigning capabilities to each of the agents. The first step can also be decomposed as follows: i) defining the overall architecture, ii) including new actors for delegation of subgoals on goal analysis of system's goals, iii) including new actors according to the choice of a specific architectural style, iv) including new actors which contribute positively to the fulfillment of some non-functional requirements.

Figure 7 depicts the initial version of our proposed system's architectural model when we execute the first step of the architectural design phase. In order to delegate subgoals of the system to the actors exactly, we added four new actors to the system: Financial Analysis Manager, SIS Manager, Order Manager and Bid/Ask Manager.

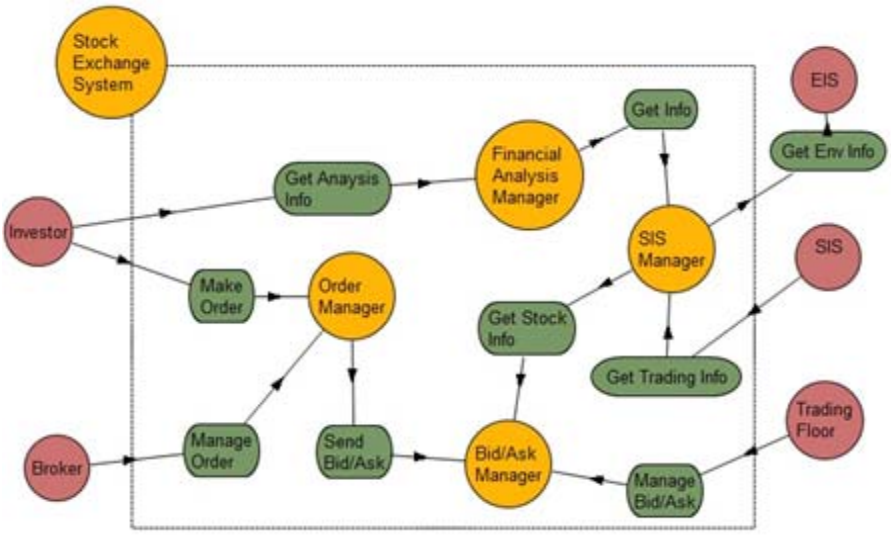


Figure 7: Architectural design (step 1)

For step 2, each actor's capabilities can be identified with the aid of the extended actor diagram (see open balloons in Figure 8), since each dependency relationship can give place to one or more capabilities triggered by external events.

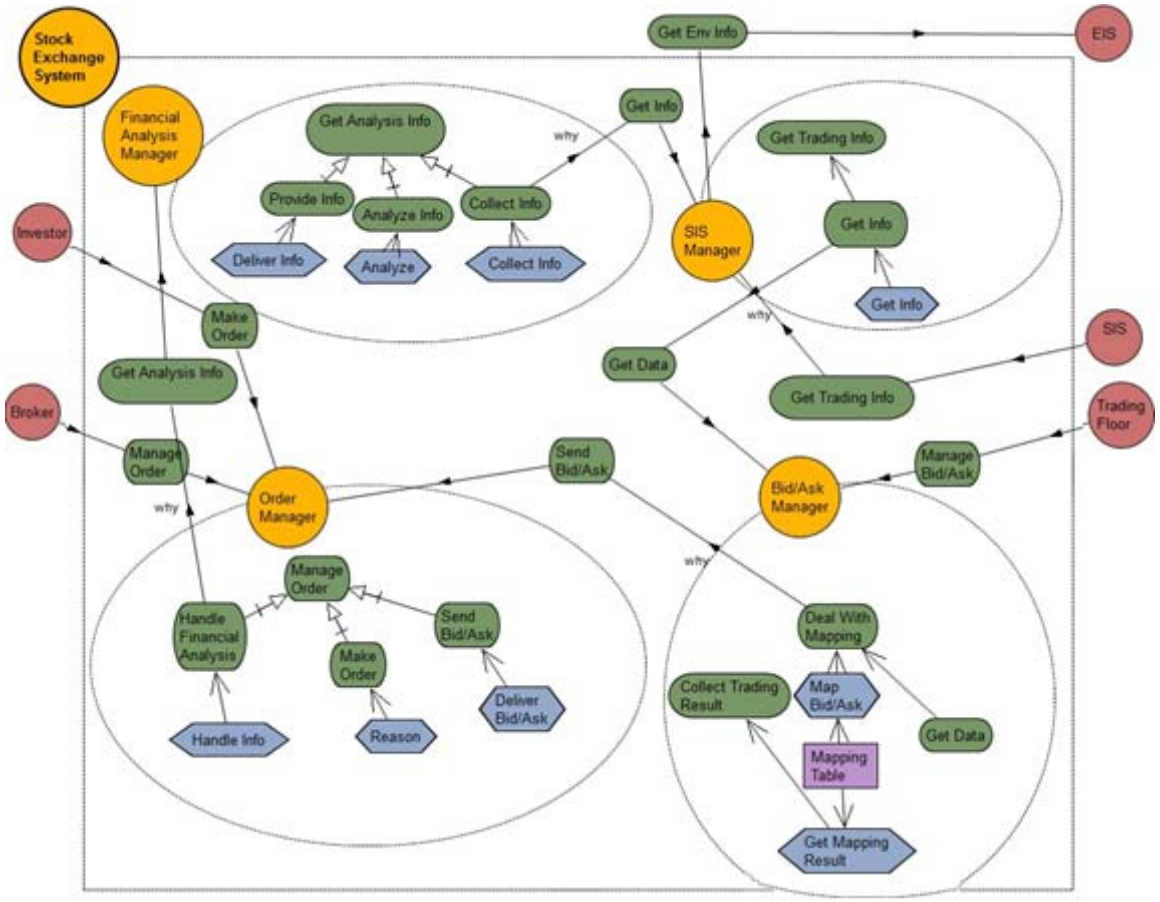


Figure 8: Architectural design (step 2)

Therefore, SIS Manager has four capabilities: Get Analysis Info, Get Trading Info, Get Stock Info and Get Environment Info. For example, in order to provide the financial information to the "Financial Analysis Manager", the "Get Analysis Info" capability is needed. At step 3 of the architectural design phase, we assigned capabilities to all agents (actors) and showed them in Table 6. As we can see in Table 6, "Bid/Ask Manager", as one of the main actors, is responsible for providing Bid or Ask and also responsible for stock info.

During determination of agent capabilities, we follow the approach given in [28]. According to this approach, capabilities consist of two parts: ability and opportunity. Ability part corresponds to a plan of a goal and opportunity part corresponds to a softgoal to which the plan contributes. For example, as depicted in Figure 8, "Financial Analysis Manager" agent

has three capabilities (Cap3, Cap4 and Cap5 in Table 6) which consist of three goals "Provide Info", "Analyze Info", and "*Collect Info*" and three corresponding plans "Deliver Info", "Analyze", and "Collect Info" respectively. Thus, the ability part of the "Financial Analysis Manager"'s capability is composed of three plans, whereas this agent has no opportunities (softgoals).

Table 6: A portion of the actor capabilities in the system

Agent	***Capability***	***Means_End(goal,plan)***	***List of contributions***
Bid/Ask Manager	Cap1	Deal With Mapping,Map Bid/Ask	{null}
Bid/Ask Manager	Cap2	Collect Trading Result,Get Mapping Result	{null}
Financial Analysis Manager	Cap3	Provide Info,Deliver Info	{null}
Financial Analysis Manager	Cap4	Analyze Info,Analyze	{null}
Financial Analysis Manager	Cap5	Collect Info,Collect Info	{null}
Investor	Cap6	Make Investment Decision,Determine Investment Condition	{null}
Investor	Cap7	Determine Strategy,Get Stock Info	{partner's interest +}, {need for liquity +}
Investor	Cap8	Make Investment Decision,Get Stock Info	{partner's interest +}, {need for liquity +}
Order Manager	Cap9	Manage Order,Reason	{null}
Order Manager	Cap10	Handle Financial Analysis,Handle Info	{null}
Order Manager	Cap11	Manage Order,Deliver Bid/Ask	{null}
Order Manager	Cap12	Send Bid/Ask,Deliver Bid/Ask	{null}
Order Manager	Cap13	Make Order,Reason	{null}
SIS	Cap14	Info Monitor And Manipulation,Analytical Process	{collect up-to-date data +}
SIS	Cap15	Info Monitor And Manipulation,Get Stock Data	{collect up-to-date data +}
SIS	Cap16	Info Monitor And Manipulation,Event Collection	{collect up-to-date data +}
SIS Manager	Cap17	Get Info,Get Info	{null}
Stock Exchange System	Cap18	Execute Order,Gather Stock Type	{secure and fast information retrieval +}
Stock Exchange System	Cap19	Execute Order,Gather Lot Amount	{secure and fast information retrieval +}
Stock Exchange System	Cap20	Execute Order,Gather Time Interval	{secure and fast information retrieval +}
Stock Exchange System	Cap21	Execute Order,Gather Price Scope	{secure and fast information retrieval +}
Stock Exchange System	Cap22	Update Stock Info,Collect Stock Info	{search for secure and real-time data +}
Stock Exchange System	Cap23	Send Bid/Ask,Get Bid/Ask	{fault-tolerance +}
Stock Exchange System	Cap24	Send Bid/Ask,Delivery Bid/Ask	{fast_delivery +}, {fault-tolerance +}

It is also worth mentioning that capability table can be automatically generated with the help of TAOM4E tool by right clicking on a system model and choosing “Show capability Table”. Table 6 shows only a portion of the capability table of our system due to space limitations. In this table, there are four columns and several rows corresponding to agents. *Means_End(goal,plan)* column represents means-ends relationship between plans and goals. In other words, it is related to ability part of the corresponding capability. *List of contributions* column represents softgoals regarding to corresponding means-end relationship (ability). Note that, capability table construction starts from early requirements phase.

4.4. Detailed design phase

In this phase, the behaviour of each architectural component is defined in further detail. Each agent is specified at the micro-level. That means we model capabilities and plans within UML activity diagrams and interactions between agents within Agent Unified Modeling Language (AUML) [29] sequence diagrams.

An UML activity diagram allows us to model a capability (or a set of correlated capabilities) from the perspective of a specific agent. Each plan node of a capability diagram can be further specified by other UML activity diagrams. We aggregate both diagrams in one diagram. For instance, Figure 9 shows the plan diagram of the “Make Order” capability. According to this figure, to give an order, first the financial analysis info is requested and manipulated. Then an order is made according to them. At the next step, the order is checked. If it is OK, a Bid will be made. If not, the order will be finalized. After making a Bid, it will be inserted into the mapping table. Then the table is examined for a possible mapped pair. If there is no case, the trade will be reported as failed, but if there is at least one, the trade will be notified to the participants, the resulting info will be saved, the trade will be reported as successful and the order will be finalized. In this way, when financial analysis info is requested, sequence of actions are executed and ended with finalize order action.

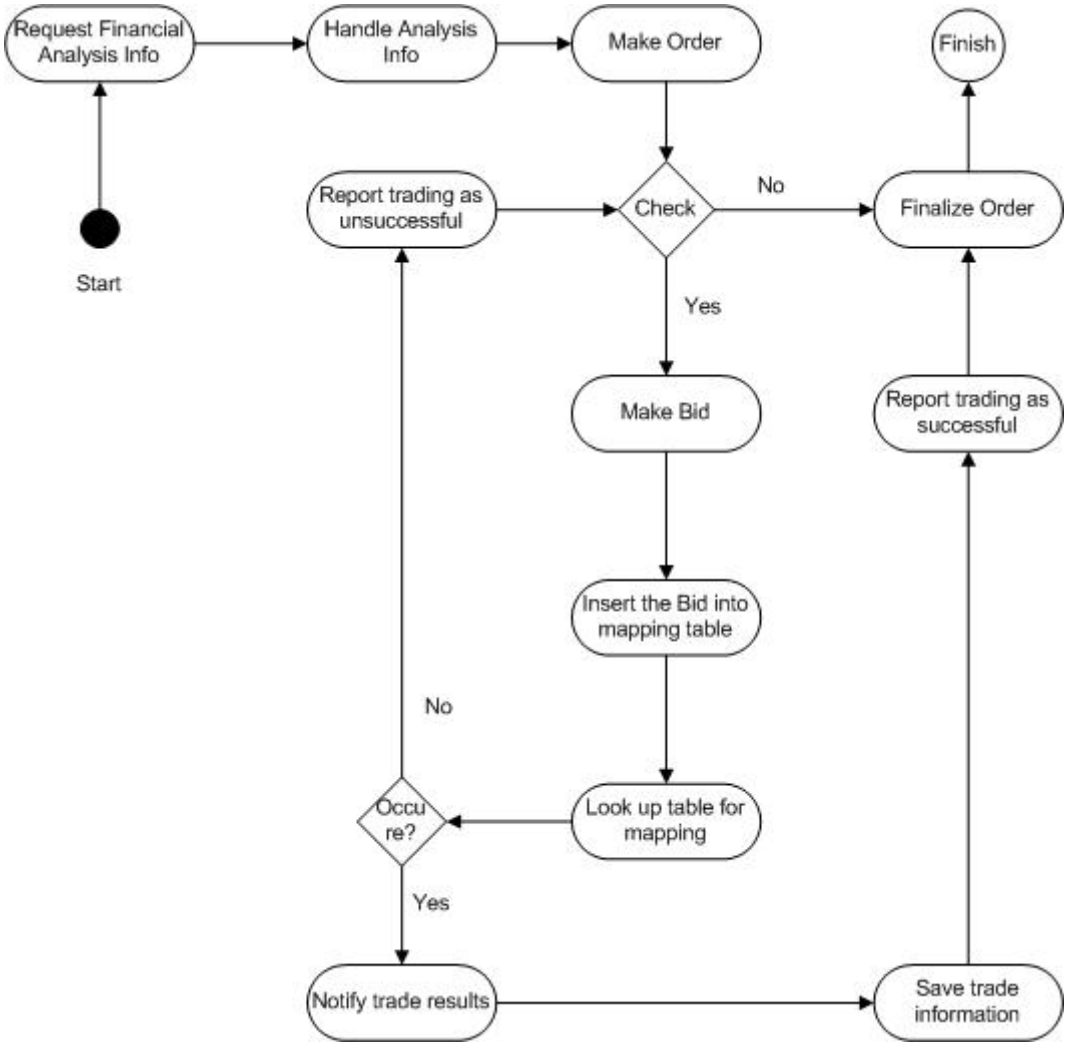


Figure 9: Activity diagram for the "Make Order" capability

In AUML sequence diagrams, agents correspond to objects (whose lifeline is independent from the specific interaction to be modeled) and communication acts between agents correspond to asynchronous message arcs. For example, Figure 10 exhibits the interactions amongst buyer agent, order manager, bid/ask manager, financial analysis manager and stock information service agent. Different than other participants, Buyer is symbolized with a stick figure since he/she is the main actor during interactions.

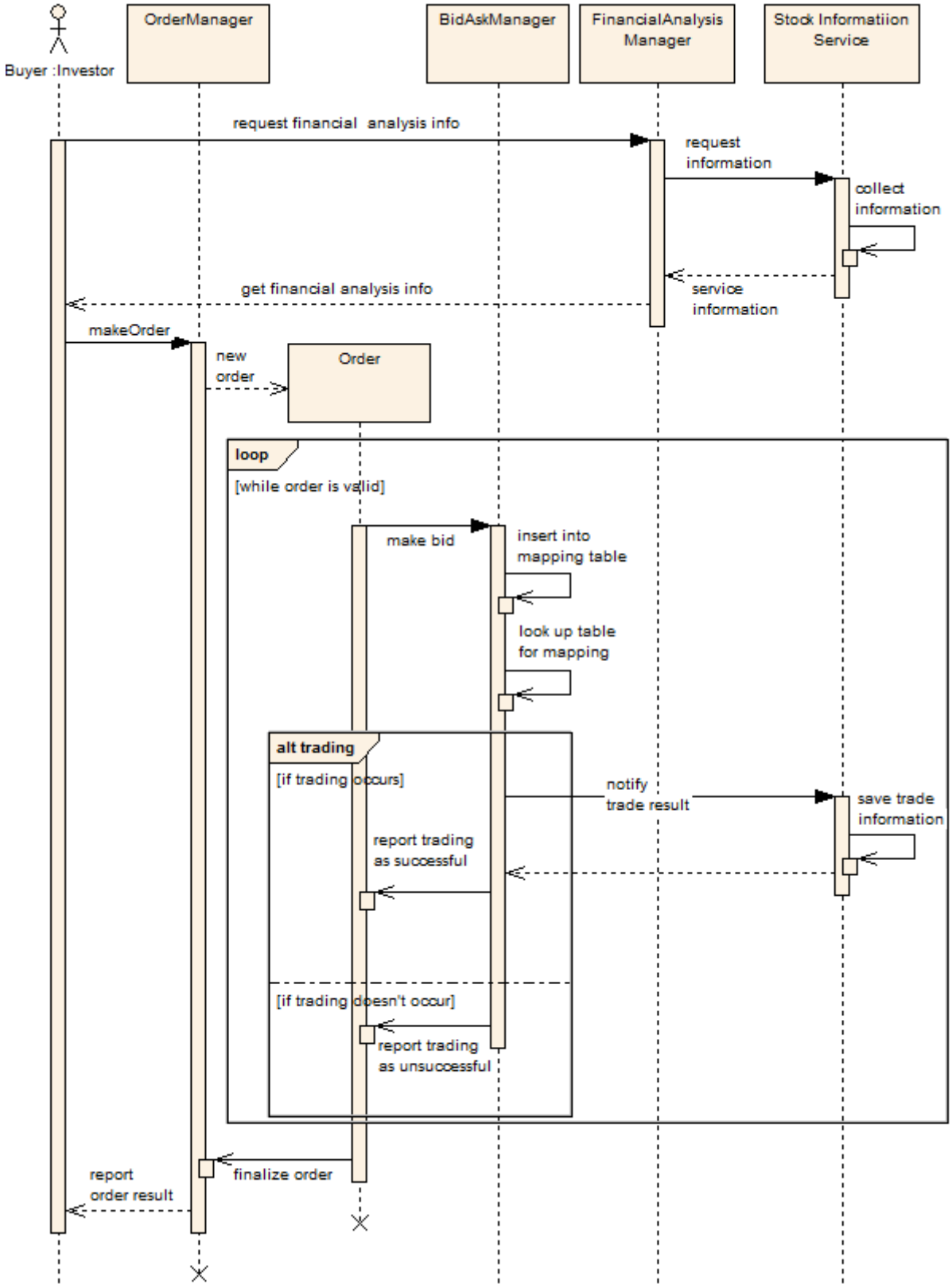


Figure 10: AUML sequence diagram for the interactions amongst buyer agent, order manager, bid/ask manager, financial analysis manager and stock information service

4.5. Implementation of the MAS

After completion of analysis and design of the system using Tropos methodology, next is the implementation of the MAS based on a real agent framework. As discussed in Section 3, JADEX framework [30] is chosen for the realization of the system and our designed agents are implemented as JADEX BDI agents.

The implementation phase of the system is composed of two steps: at first, TAOM4E tool [24] is employed for the code generation. Design models gained from the analysis and design phases are utilized within a model-to-text transformation and template codes for agent beliefs, desires and plans are automatically generated. For instance, we use TAOM4E's "t2x tool" to generate templates for JADEX ADF files of system agents. All elements of an ADF can be automatically created with this tool and later modified by system developers. These auto-generated ADF elements are imports, capabilities, beliefs, goals, plans, events, expressions, properties and configurations. However, that code generation is not enough for the exact implementation of the system. So, in the second step, generated codes are completed manually.

JADEX is based on JADE [31] and it implements BDI architecture, including agent's plans, goals and beliefs, over JADE structure. Therefore, in higher level we implement plans, goals and facts of beliefs for agents and in lower level JADEX implements them using JADE's normal agents and different types of behaviors defined according to the behavior library of the JADE API.

There are three major notions regarding to JADEX architecture: beliefs, goals and plans. Beliefs are Java objects which represents the environmental facts that an agent have and are stored in a beliefbase. The beliefbase contains the facts that an agent possesses. In other words, it represents the knowledge of the world in which the agent is situated. Beliefs may change in course of time in a dynamic environment, thus beliefbase need to be updated in the long run. Goals in JADEX resemble desires mentioned in [11] to some extent. However, goals are vital part of JADEX rather than events in traditional BDI systems. Detailed explanation of goals and goal types are previously discussed in Section 3. Finally, plans are Java classes that are executed in order to achieve a goal of an agent. Plans have two parts: plan head and plan body. A plan head can be automatically generated by TAOM4E tool however plan body should be coded by the programmer.

Considering the JADEX structure, each agent should have an ADF which is an XML formatted file and codes of related plans in Java language for that agent. An ADF describes the structure of an agent. In other words, ADF defines the agent's elements. It defines the capabilities of an agent including beliefs, goals and plans. Belief set is composed of fact variables which will be used with plans of agents. ADF files use reference names for elements to be referred within plans.

To give some flavour of the implementation, BDI definition, plan generation and body of a critical plan of the BidAskManager agent are discussed in this paper. BidAskManager is chosen since it is one of the featured agents in our stock trading system. The other agents mostly do the normal negotiation, communication and interaction with each other. In addition to those tasks, BidAskManager deals with request for Bid and Ask along with gathering stock info and updates its beliefs to have the highest ability to make the best decisions at any time.

The ADF for BidAskManager agent, given in Listing 1, describes agent's structure including capabilities, goals and configurations. In the ADF, at the first step, the agent's name (Bid-Ask-Manager in here) and its package are defined. Then necessary imports from different libraries are done. For example, JADEX plan library is imported. Secondly, agent's capabilities are defined. These capabilities include plans, goals, events, beliefs, and configurations. According to JADEX structure, capabilities can be defined both in ADF and CDF (Capabilities Definition File). Both of them can include all of the items of a capability mentioned before. However, the capabilities which are defined in an ADF are purely dedicated to that agent while those which are described in CDF belong to all of the agents. In our implementation, some of the capabilities, which are shared among all of the agents, are defined in a CDF named *StartAgents* and added to BidAskManager using a local name of *"startcap"*. Also another predefined CDF from JADEX is added to this agent, namely *AMS* which is a CDF in *jade.planlib* path. This has a local name of *amscap* and is used to access agent management capability in the system. Both of these common CDFs, StartAgents (special for our implementation) and AMS (for whole JADEX structure), are added between *<capabilities>* tag. Nevertheless, our agent has even more capabilities which are special for it and defined in the rest of the ADF. Definition of the capabilities will be shown in the remaining listings (2 - 6) and are discussed below.

```
<agent name="Bid-Ask-Manager" package="jadex.Projects.stockexchange.bid-ask-manager">
   <imports>
        <import>jadex.planlib.*</import>
        <import>java.util.*</import>
   </imports>
   <capabilities>
        <capability name="startcap" file="StartAgents"/>
        <capability name="amscap" file="jadex.planlib.AMS"/>
   </capabilities>
   <goals>
            <!--  see Listing 2 -->
    </goals>
    <beliefs>
            <!--  see Listing 3 -->
    </beliefs>
    <plans>
            <!--  see Listing 4 -->
    </plans>
    <configurations>
            <!--  see Listing 5 -->
    </configurations>
    <events>
            <!--  see Listing 6 -->
    </events>
</agents>
```

Listing 1: Structure of the ADF for BidAskManager BDI agent

Listing 2 shows the part of the ADF in which agent goals are declared. In our implementation, the first goal which is added to agent's goals is *"Dealing with mapping"*. This goal has two parameters, company name and goods which is offered, both in "in" direction. This goal aims to catch the lowest price if the goods list is not empty. As the next goal, the agent has its main goal, *"Get data"*. This goal will be achieved when the agent's data is not updated. Then the goal references are declared to be accessed by the agent's plans. According to JADEX specification, goal definition for agent startup and termination should also be inserted. Hence at the end of the listing, declaration of the "start_agents" and "ams_destroy_agent" goals take place.

```
<achievegoal name="Deal with mapping">
            <parameter name="Company" class="String" direction = "in" />
            <parameter name="Goods" class="String" direction = "in" />
            <targetcondition>$goal.price==$goal.lowest</targetcondition>
            <creationcondition> $beliefbase.goods!=$null</creationcondition>
</achievegoal>
<maintaingoal  name="Get data" exclude="when_is_update">
            <maintaincondition>$beliefbase.info.state != $update  </maintaincondition>
</maintaingoal>
<achievegoalref name="start_agents">
            <concrete ref="startcap.start_agents"/>
        <deliberation>
            <inhibits ref="ams_destroy_agent"/>
        </deliberation>
</achievegoalref>
<achievegoalref name="ams_destroy_agent">
             <concrete ref="amscap.ams_destroy_agent"/>
</achievegoalref>
```

Listing 2: Goals of the BidAskManager agent

The next part in our agent's definition include beliefs (see Listing 3) in which the related references, belief sets and their facts are declared. First, a belief set reference, namely *info*, is declared for exchange data in the system. This is utilized in the related plans to access the necessary facts existed in aforementioned agent's beliefs. Next, a fact named *update* as a separated belief (named *state*) is declared. The belief is in type Boolean and keeps the current state of the beliefs of agent whether they are updated or not. Finally, part of the agent's facts entitled *TradeRules* is declared.

Actually these facts represent the system's decision principles. Each fact in this section is an instance of Tuple class and keeps the relation of financial effect of one good to another along with the amount of its effect in real world. For example, the first tuple ("gold", "petrol", -90) means that increasing the price of gold in the market can affect approximately 90% on the price of petrol but in reverse direction.

```
<beliefsetref name="info">
        <concrete ref="exchange.data" />
</beliefsetref>
<belief  name="state" class="boolean">
        <fact>update</fact>
</belief>
<beliefset name=" TradeRules " class="Tuple" exported="true">
        <fact>new Tuple("gold", " petrol", -90)</fact>
        <fact>new Tuple("Iron", " gold", 30)</fact>
        <fact>new Tuple("War", " Petrol", 50)</fact>
</beliefset>
```

Listing 3: Beliefs of the BidAskManager agent

As we can see in Listing 4, there are plans at the rest of our agent's ADF. Each plan is in fact the live part of the agent when it is running. The plans are working when they are active. Thus, they can change the state of the agent by updating, creating, manipulating, activating or omitting a goal, event, another plan, beliefs or facts. They also can react for the messages from other plans in current agent or other agents in the same platform or even from other platforms.

The first plan in Listing 4 is named *MapBidOrAsk* with highest priority. Its Java class name is *MapBidOrAsk* which in fact constitutes the body of the plan. This plan has a parameter named *BidOrAsk* and its type is *Request* class. Also the reference needed in plan code is described here as *"request_sellOrbuy.content"* for message event mapping. After that, a reference is

defined for wait queue, called *request_sellOrbuy*. Then the trigger requited for the plan is defined. The goal required to trigger this plan is *"Deal_with_mapping"* in which a matching for coming event will be done. This coming message should be *Ask* and its direction should be towards the agent's plan.

In the next step, the binding required for the plan is done. The expressions in this section are inserted between <bindings> tags and written in OQL (Object Query Language) [32] which is very similar to SQL. By this binding, the plan can find the goods in the system which are denoted in the arrival message (see the <binding> called *"find"* in Listing 4).

The second plan, named *GetResult* has the body class *Result* which gets a parameter (named *BidOrAsk*) in type of *Reply* class. This plan's goal mapping reference is *"exchange.trade"*. Also, the plan's trigger has a message event reference named *reply_sellOrbuy* and a condition of not to have a null trade in system's beliefs at runtime.

```
<plans>
      <plan name="MapBidOrAsk" priority="1">
         <body class="MapBidOrAsk"/>
         <parameter name="BidOrAsk" class="Request">
            <messageeventmapping ref="request_sellOrbuy.content"/>
         </parameter>
         <waitqueue>
            <messageevent ref="request_sellOrbuy"/>
         </waitqueue>
         <trigger>
            <messageevent ref="request_sellOrbuy"/>
             <goal ref="Deal_with_mapping">
               <match>$event.equal("Ask")</match>
                <parameter ref="direction">
                  <value>inside</value>
                </parameter>
             </goal>
         </trigger>
         <precondition>  MapBidorAsk.containsGoods((String)$event.content)  </precondition>
         <bindings>
             <binding name="find">
                 select $goods from $beliefbase.info where $goods.getName().equals($event.goal.goods)
            </binding>
         </bindings>
      </plan>
      <plan name="GetResult">
         <body> new Result ( ) </body>
         <parameter name="BidOrAsk" class="Reply">
            <goalmapping ref="exchange.trade"/>
         </parameter>
         <trigger>
            <messageevent ref="reply_sellOrbuy"/>
            <condition>$beliefbase.trade!=null</condition>
         </trigger>
       </plan>
</plans>
```

Listing 4: Plans of the BidAskManager agent

Another important part of an ADF is BDI Agent's configuration. Listing 5 shows the configuration of BidAskManager agent. Default configuration, in here, defines the initializations needed for system goals (*start_agents* and *ams_destroy agent*). Also the ADF defines the initializations needed for plans of the system. Initialization of the in-goal *ams_destroy_agent* (given in the bottom of Listing 5) has a parameter reference named *agentidentifier* and its value is agent's ID.

On the other hand, initialization of *start_agents* has the parameter reference *agentinfos* and includes two values (see the upper part of Listing 5). The first value, representing a buyer to propose some bids, is an instance object from *StartAgentInfo* class including *Buyer*, name of the buyer (*Morey*), and an object including some *Bid* objects proposing the companies and firms which the buyer is interested in them. In here four bids are prepared for the stocks of TCELL (Turk Cell: a telecommunication company in Turkey), AKBNK (Ak Bank: a bank in Turkey), GSRAY (Galatasaray SK: a football club in Turkey) and ISBTR (Is Bank: another bank in Turkey) along with their stock information. For instance the first Bid object is initialized with the parameters *Date(System.currentTimeMillis()+360000), 8.45, 8.60* and *100*. That means the agent will have a bid on TCELL stock which expires in 1 hour. Minimum stock prize is 8.45 and agent can increase it to 8.60 at maximum. Total amount of lots, that our agent needs to buy, is given as 100 for that bid.

The second value is similar to the first one but it is for *Seller* who is interested in selling some shares from specified companies and firms for mentioned fees. As a configuration for plans of the system, ADF defines *MapBidOrAsk* plan for initial plan of agent.

```
<configuration name="default">
   <goals>
      <initialgoal  ref="start_agents">
         <parameterset ref="agentinfos">
            <value>
               new StartAgentInfo("jadex.examples.stockexchange.investor.Buyer", "Morey", 0, new String[]{"initial_bids"},
                  new Object[]{new Bid[] {
                     new Bid("TCELL", new Date(System.currentTimeMillis()+360000), 8.45, 8.60, 100),
                     new Bid("AKBNK", new Date(System.currentTimeMillis()+360000), 7.80, 7.90, 150),
                     new Bid("GSRAY", new Date(System.currentTimeMillis()+360000), 175, 179, 5 ),
                     new Bid("ISBTR", new Date(System.currentTimeMillis()+60000), 2000, 2140, 2 )
                  }})
            </value>
            <value>
               new StartAgentInfo("jadex.examples.stockexchange.investor.Seller", "Sulo", 0, new String[]{"initial_asks"},
                  new Object[]{new Ask[]{
                     new Ask("TCELL", new Date(System.currentTimeMillis()+360000), 8.60, 8.65, 200),
                     new Ask("AKBNK", new Date(System.currentTimeMillis()+360000), 7.92, 7.98, 100),
                     new Ask("GSRAY", new Date(System.currentTimeMillis()+360000), 177, 183, 5),
                     new Ask("ISBTR", new Date(System.currentTimeMillis()+60000), 2135, 2145, 2)
                  }})
             </value>
            </parameterset>
         </initialgoal>
         <initialgoal ref="ams_destroy_agent">
            <parameter ref="agentidentifier">
               <value>$agent.getAgentIdentifier()</value>
            </parameter>
         </initialgoal>
   </goals>
   <plan>
      <initialplan  ref =”MapBidOrAsk” />
   </plan>
</configuration>
```

Listing 5: Configuration of the BidAskManager agent

The last part of the ADF of our BDI agent covers the events (Listing 6). In this part, the events or messages which will be exchanged between the agent's own plans or other agents are declared. These messages can be either external or internal. In our implementation, there are two external messages namely *request_sellOrbuy* and *reply_sellOrbuy*. The former comes from the outside environment of the agent to the agent and it is a FIPA REQUEST message [33]. The latter one is an outgoing message from other agents and it is a FIPA REPLY type message [33]. Also there is one possible internal event which is called *gui_update*. This event will be triggered by the GUI components and will lead to an internal event and in result agent’s plan will have proper reaction to that event.

```
<messageevent name="request_sellOrbuy" direction="receive" type="fipa">
    <parameter name="performative" class="String" direction="fixed">
        <value>jadex.adapter.fipa.SFipa.REQUEST</value>
    </parameter>
</messageevent>
<messageevent name="reply_sellOrbuy" direction=”send” type="fipa">
    <parameter name="performative" class="String" direction="fixed">
        <value>SFipa.REPLY</value>
    </parameter>
</messageevent>
<internalevent name="gui_update">
    <parameter name="content" class="String[]"/>
</internalevent>
```

Listing 6: Events of the BidAskManager agent

Finally, we discuss sample codes from BidAskManager agent's plan body, in order to exemplify how processing steps are implemented. Listing 7 includes Java class written for the *MapBidOrAsk* plan of BidAskManager agent. As we may remember, header of this plan has been previously given as the first plan in ADF of BidAskManager agent (see Listing 4). MapBidOrAsk extends jadex.runtime.Plan class and each instance of this class starts to execute from its *body()* method as expected from a JADEX plan. When a make bid/ask event message comes from Order Manager, the message is parsed by *getParameter()* method and the order is extracted by a method called *ExtractOrder()* to understand whether this is a selling or buying order. Then the order is inserted to the open order file as mentioned in Section 2. For demonstration purposes, we initialized the table with several stocks like TCELL, AKBNK and used a java.util.Vector object to add new orders during execution of the system. After inserting the order to the table, stock exchange system checks whether an order, e.g. selling order, is matched with corresponding order, e.g. buying order inside *isAnyMatch()* method. Note that this method applies the multiple price-continuous auction rules in order to check matching. If there is a match, transaction is executed by *doTransaction()* method and GUI for representing the passive file order is updated. Otherwise, a new goal is created for this unmatched order via *createGoal()* method. If *MapBidOrAsk* plan is successful, *passed()* method is called and beliefbase of the agent is updated. Successful transaction event message is also sent to both Order Manager and SIS Manager as previously depicted in Figure 10. If *MapBidOrAsk* plan is failed, *failed()* method is called and unsuccessful transaction event message is sent to Order Manager.

```
import java.util.*;
import jadex.runtime.*;
import java.util.StringTokenizer;
public class MapBidOrAsk extends Plan {
    public MapBidOrAsk() {     // Constructor code
        getLogger().info("Created: "+ this);
    }
    // Plan attributes
    protected static Map StockTable;
    protected AgentIdentifier aid;
    protected boolean  checkTransaction;
    protected IGoal g;
    class Table {
        String StockName;
        String Type;
        int OrderNo;
        int LotQuantity;
        double price;
        Time time;
    }
    Vector<Table> table = new Vector<Table>();
    static {                            // static block
        StockTable = new HashMap();
        StockTable.put("TCELL");
        StockTable.put("AKBNK ");
        StockTable.put("TurkTelecom");
        StockTable.put("GSRAY ");
        StockTable.put("ISBTR");
    }
    public void body() {          // Plan main code
        while(aid){
            IMessageEvent me = waitForMessageEvent("make_Bid_Or_Ask");
             String order = (String)getParameter("order").getValue();
             Order O = ExtractOrder (order);
             InsertOrder (table, O);           // Keeps the order of table according to the price
             checkTransaction = isAnyMatch (table, Order);
             if (checkTransaction == true){
                doTransaction();
                IInternalEvent event = waitForInternalEvent("gui_update");
             }
             else {
                  if ( Order.Type.equal("Sell") )  g = createGoal("sell");
                else if ( Order.Type.equal("buy") )  g = createGoal("buy");
                     try{
                         dispatchSubgoalAndWait(g);
                     }catch(GoalFailureException gfe){ /* Exception handling */  }
            }
        }
    }
    public void passed() {
        UpDate ( (Map)getBeliefbase().getBelief("----").getFact() );
        sendMessage(((IMessageEvent)getInitialEvent()).createReply("Successfull", Order_Manager));
        sendMessage(((IMessageEvent)getInitialEvent()).createReply("Successfull", SIS_Manager));
    }
    public void failure() {
       sendMessage(((IMessageEvent)getInitialEvent()).createReply("Unsuccessfull", OrderManager));
    }
}
```

Listing 7: Body of MapBidOrAsk plan

5. Evaluation

Evaluation of the study discussed in this paper is twofold. First part of the evaluation covers the lessons learned during design and development of a multi-agent stock exchange system within the software engineering perspective while second part covers the experimental evaluation of the implemented multi-agent stock exchange system and aims to reveal pros and cons of the system.

Social nature of the stock trading causes us to follow an AOSE methodology and develop the system according to the agent paradigm. Autonomous agents interacting with each other in a MAS both enhance modeling of the system and make the stock exchange software system more realistic. We are aware that AOSE is distinct from traditional object-orientation where agent, goal, role, organization, context and messages are considered as first class entities and agent-oriented development brings out a more complex and relatively time consuming development process when we consider an approach based on the object-orientation. Besides, developers coming from object-oriented paradigm may encounter some difficulties with respect to notion of agent and mentalist notions. However, we experienced that a feasible stock trading system requires much more than just simple message transferring between components and agent-based approach is the right choice considering behaviour of the traders, autonomous interaction between the traders and their inference mechanisms.

On the other hand, we found Tropos as an adequate methodology for developing agent-oriented systems in which notions of agent, goal, task and social dependency are used. It starts from very beginning (early requirements) and with the help of new tools, it accompanies agent developers to the end of code generation. Uses of the goal, plan and capability notions within Tropos also makes it highly suitable with BDI architecture.

Although Tropos covers complete agent-oriented software process, there exists some gap between its model and its exact application. For instance, we faced a problem in detailed design phase; there isn't any specification about usage of AUML activity diagram for agent capabilities (capability diagrams). We couldn't define any capability diagrams separately. Therefore, we merged capability diagrams with the plan diagrams (e.g. "Order Manager Agent"'s capability and plans are merged and it is called "Make Order"). Furthermore, software development process within Tropos does not support iterative approach. We believe

that it is a major drawback which causes difficulties related to the possible return back situations encountered in most of the agent software development processes.

Use of JADEX BDI Agent architecture allowed us to develop goal oriented agents including their beliefs, goals and plans. Intelligent agents may actively decide for themselves to execute some actions (e.g. make a bid). Therefore, our multi-agent stock exchange system converges to the real life system.

Another advantage of the utilization of JADEX framework is the enhancement of our trader agent capabilities with Reactive Reasoning and Planning computational model. Agent systems conforming to this model are event-based. When a new event is generated, an agent makes meta-level reasoning and chooses the best plan to deal with it. Agents do not execute a plan if there is not an incoming message. However, a goal which is related to a plan is a key component of JADEX. We explicitly specified the goals of trader agents. JADEX represents beliefs of agents as Java objects and stores them in the belief-base, that is, the database of agent beliefs. Use of JADEX also allowed us to specify agent plans those have to be executed when a belief is changed.

During the development process we used TAOM4E computer aided design tool. Based on the Eclipse Platform [27], it offers a flexible solution to the problem of component integration. We experienced many advantages of TAOM4E. First, we are able to change Tropos meta-model to keep several Tropos dialects. Nodes (i.e. actors) or relations (i.e. dependencies) can be added or removed from the meta-model easily. Different notations can be added to existing nodes and relations. Second, TAOM4E enables the agent designer to edit models. For instance, one may create a model with visual editor, query a model on subject of some properties (i.e. number of some elements or consistency) and have different views on that model. Third, all Tropos phases are supported by TAOM4E and user is guided during Tropos process. The tool provides some clues and analysis choices during agent-oriented software development. We also experienced good and essential limitations during development process. For example, TAOM4E does not let designer link an agent or actor directly to a resource because it should be done by only mediation of a goal. Last but not least, it provides integration with other tools. It may translate its own model to other languages like UML.

In addition, TAOM4E enables us to use many visual modeling features like drag and drop objects form outline tree to the editor, different color representations for objects (actors, resources and so on), changing the type of the objects in the editor (like hard-goal to soft-goal) and saving the diagrams as pictures and print them.

However, we encountered problems during development process as there is not enough TAOM4E documentation which tells the process step-by-step. This may lead open-ended solutions.

Considering the second part of the evaluation, we can say that this study has brought some benefits for stock exchange systems especially ISE. The study uses agents in a MAS for dealing with the problems of decisions in a virtual stock exchange system. Moreover, the work discussed in here presents an agent system with BDI architecture for stock exchange while most of the previous studies used just the reactive architecture which offers less intelligence or in some rare case studies they used Fuzzy Expert Systems (FES) to add intelligence to their system.

In our proposed system, it is possible to simulate a stock exchange and analyze the outcome of different financial plans which makes it possible to select the best and most appropriate one for business. This is simply possible by giving random inputs and events as is in the environment as well as the plans which are desired to be examined and as a result one can observe the amount of benefit gained from the plan. In this way the best plan for current real situation can be obtained.

On the other hand, it is possible to use the proposed system as a real implementation of a stock exchange and give the control of central processing, information gathering and analysis to the software in order to deal with it. Meanwhile users can work with virtual brokers of the software and take advantages of their consultant to decide what to buy or sell and how much to buy or sell to gain the most benefit. Therefore a human being does not need to deal with communication, negotiation or even planning for financial affairs. He/she just needs to give basic facts and his/her goals and intelligent system will take the best decision and plan for him/her. This plan library can re-enforce itself using plan combination, modification and update according to success or failure situations encountered in previous tries of the same plan. For example, after accomplishment of a plan and gaining bad results, plan library can

either remove the plan or update it to be ready for next use and give a better result. In this way the plan library will be re-enforced during the whole execution.

6. Related Work

Various studies exist which use agents for stock exchange. As one of the early studies, El-Sawi and Ali [2] propose mobile agents for realizing stock exchange. Buyer and seller agents are named as "traders". Stocks are virtual objects which are constructed as persistent Java classes. In order to simulate the market, a software entity called, Virtual Stock Exchange which consists of Verification (registration), Transaction Server (the place where agents post their intent for a trade) and Information Server (stores information about stocks) components, is introduced.

Based on the study in [4], Kendall and Su simulate a stock market where trader agents get some predefined indicators. Indicators (also discussed in [8]) correspond to the beliefs of agents. If an agent gets profit it publishes its strategy to the strategy pool for the use of other agents. In this study, 50 artificial traders were generated randomly. They traded a stock in a simulated stock market, learned to trade by them and learned from other traders through social learning. Although 80% of artificial agents learned successfully in stock trading, authors did not consider agents' intelligence in decision process.

In Posada's study [5], software agents switch between alternative bidding strategies during Continuous Double Auction [13] unlike the previous studies. Six scenarios are simulated and compared. Each scenario involves three static environments and two kinds of agent strategic behaviors for each of these static environments. Agent behaviors are compared in the following situations: homogeneous populations with fixed learning strategies during the auction under static and dynamic environments; heterogeneous populations with fixed learning strategies during the auction under static and dynamic environments; static and dynamic environments when the agents are allowed to change their strategies during the auction for heterogeneous populations.

A middle layer agent system, situated between the demand and the supply sides of the information, is introduced in [3]. In this system each agent has partial knowledge about the system which has 3 layers: Stock Information Retrieval, Stock Status Monitoring and Buying

and Selling Shares Decision Support. Included agent types are: interface agent, coordinator agent, profiler agent, monitoring agent, communication agent, risk management agent, technical analysis agent, fundamental analysis agent, and decision-making agent.

New York Stock Exchange is implemented in [6]. Authors use TrAgent distributed architecture [9] and they focus on stock broker which is the main actor of the negotiation. Different kind of agents like investor, stock broker, floor broker and specialist make collaboration in order to meet the user's request. Investor agent is a representative of a real investor and initiator of any transaction. Stock broker agent manages the transactions for the investor agent, provides the investor agent with an ordered list of the firms in which the investor agent is interested, ranking them according to their possible profitability. Floor broker agent receives orders from stock broker agents, locates the trading post and executes the trade. Specialist agent provides methods for negotiating prices with the floor broker agent. All these events take place in trading floor. When investor agent places an order, it is forwarded to stock broker and then to floor broker who negotiates with specialist agent. If bid and ask matches, result is reported to the investor. Authors make use of JADE framework [31] to develop the multi-agent trading system. Although the study presents a MAS which converges to the real life scenario, stock brokers in the system do not predict anything about future values of firms. In addition, used indicators are static; they do not change in time.

Each of the abovementioned studies provides valuable approaches for developing stock exchange systems. For instance, agents have strategies in [8]. Reuse of agents' experiences adds more reality to the system. It is like expecting to make profit by using one's successful strategy. On the other hand, authors use belief notion in agents like [5] did. But there is no intelligence of agents during decision process in [8]. Study in [5] uses a similar approach but strategies are fixed. Authors focus on different combinations of agents and environment (e.g. same kind of agents in dynamic environment) when implementing the system. That is different from the changing strategies discussed in [8].

MASST framework [3] is a well-organized and large scale framework. There is an effective interaction between agents realizing financial tasks such as risk management and technical analysis information retrieval process. In addition, real life data are extracted from Internet, companies, and commentators. However, authors do not use beliefs in agent architecture

unlike [5] and [8]. Moreover, a task is generally executed with information gathering and filtering and reporting the result to the related agent.

In [6], an artificial intelligence technique is used and security issues are emphasized. However, financial indicators do not change in time. Also stock broker agent who is the key component for negotiation does not predict anything about future values of the firms.

Although [5] and [8] use belief notion for some of their agents, they do not totally use BDI architecture. Our agents are designed with their beliefs, plans and goals from the beginning to the end of the design. None of the abovementioned studies use JADEX as implementation tools. Implementation of these studies is done mostly with JADE framework (e.g. [3], [6]) or even details of the implementation are not given (e.g. [2], [5], [8]). We believe that decision process of agents is vital in stock exchange system. Therefore, in order to have an intelligent system, we consider every agent with Belief-Plan-Goal. Indeed, every person has some beliefs about stock he/she wants to buy, e.g. "economic crisis will not affect the company X until the first quarter of 2011". People also has plans to make profit, e.g. "I will buy stock Y when Euro is under 2.015 Turkish Liras" and goals like "I do not want to make a risk if Euro falls under 2.015 Turkish Liras".

The work presented in this paper also contributes to the related research area by discussing whole design and implementation process of a multi-agent stock exchange system. Although abovementioned studies include noteworthy proposals on agent based market simulation and/or theoretical design issues of agent based stock exchange systems, only a very few of them considers exact implementation within the software engineering perspective. The work discussed here aims to fill this gap and guide to the software engineers for constructing such software systems starting from the scratch.

7. Conclusion and Future Work

Design and implementation of a multi-agent stock exchange system[3] is discussed in this paper. From early requirements to the concrete implementation, all of the phases of the system development are completed. Trading rules and procedures of one of the important

[3] All analysis and design models and source codes for implemented BDI agents can be available at: http://ube.ege.edu.tr/~kardas/stock_exchange.zip

stock exchange systems are adopted. Determination of social actors, their dependencies and interactions between each other as a result of a deep means-ends analysis paves the way of the development of a MAS which aims to provide a realistic model of the traders. In this study, we show how facts about the current stock market can be represented as agent beliefs, how goal of each stock trader can be modeled as agent desires and how intentions (in other words agent plans) can be constructed in order to achieve goals of stock traders. Each participant in the system is first designed as BDI agents with their beliefs, goals and plans and then BDI reasoning and behavioural structure of the designed agents are exactly implemented.

Our next work is to develop mobile versions of the implemented agents in order to use the system in various mobile devices. Another planned work is to add ontology capability and inference mechanism to the system as it can use the information among many stock exchange systems all over the word. Hence, we can provide real plans for special financial issues to test and compare the results with real systems.

References

[1] Wooldridge M, Jennings NR. Intelligent agents: theory and practice. *The Knowledge Engineering Review* 1995; **10** (2): 115-152. DOI: 10.1017/S0269888900008122.

[2] El-Sawi K, Ali D. Mobile Agent Technology for Dynamic Stock Exchange, 1998 International Conference on Industry, Engineering, and Management Systems (IEMS '98), Cocoa Beach, FL, U.S.A. March 1998.

[3] Luo Y, Liu K, Davis DN. A multi-agent decision support system for stock trading. *IEEE Network* 2002; **16** (1): 20-27. DOI: 10.1109/65.980541.

[4] Kendall G, Su Y. The Co-evolution of Trading Strategies in a Multi-agent based Simulated Stock Market through the Integration of Individual Learning and Social Learning. 2003 International Conference on Machine Learning and Applications (ICMLA'03), Los Angeles, U.S.A. June 2003; 200-206.

[5] Posada M. Strategic Software Agents in Continuous Double Auction Under Dynamic Environments, *Lecture Notes in Computer Science* 2006; **4224**: 1223-1233. DOI: 10.1007/11875581_145.

[6] Rahimi S, Tatikunta R, Ahmad R, Gupta B. A multi-agent framework for stock trading. *International Journal of Intelligent Information and Database Systems* 2009; **3** (2): 203–227. DOI: 10.1504/IJIIDS.2009.025163.

[7] Sycara KP. Multiagent Systems. *AI Magazine* 1998; **19** (4): 79-92.

[8] Kendall G, Su Y. A multi-agent based simulated stock market - testing on different types of stocks, 2003 Congress on Evolutionary Computation (CEC'03), December 2003; 2298-2305.

[9] Bjursell J, Rahimi S, Wang CF. Modelling the stock exchange market using mobile agent technology and its security issues. *International Journal of Education and Information Technology* 2004; **1** (1): 28–38.

[10] Rust J, Miller J, Palmer R. Behavior of trading automata in computerized double auctions. In Friedman and Rust (Eds.): The double auction markets: Institutions, theories and evidence, Addison-Wesley, 1993; 155-198.

[11] Bratman ME. *Intention, Plans, and Practical Reason*. Harvard University Press: Cambridge, Massachusetts, 1987.

[12] Istanbul Stock Exchange. http://www.ise.org [8 June 2011].

[13] Zou Y. *Agent-based Services for the Semantic Web*. Doctoral Dissertation, University of Maryland, Baltimore County, Maryland, U.S.A.; 2004.

[14] Kucukkocaoglu G. Single-Price Auction System for the Istanbul Stock Exchange. *ISE Review* 2004; **8** (29): 66-79.

[15] Istanbul Stock Exchange Guide. http://www.ise.org/Training/Guides.aspx [8 June 2011].

[16] Bresciani P, Perini A, Giorgini P, Giunchiglia F, Mylopoulos J. Tropos: An agent-oriented software development methodology. *Autonomous Agents and Multi-Agent Systems* 2004; **8** (3): 203-236. DOI: 10.1023/B:AGNT.0000018806.20944.ef.

[17] Castro J, Kolp M, Mylopoulos J. Towards requirements-driven information systems engineering: the Tropos project. *Information Systems*, 2002; **27** (6): 365-389. DOI: 10.1016/S0306-4379(02)00012-1.

[18] Yu E. *Modelling strategic relationships for process reengineering*. Doctoral Dissertation, University of Toronto, Toronto, Ont., Canada;1995.

[19] Giorgini P, Kolp M, Mylopoulos J, Castro J. (2005). Tropos: A Requirements-Driven Methodology for Agent-Oriented Software. In Henderson-Sellers, B. and Giorgini, P. (Eds.): Agent-Oriented Methodologies, Idea Group Publishing, 2005; 20-46.

[20] Rao A, Georgeff M. BDI Agents: From Theory to Practice. First International Conference on Multi-Agent Systems (ICMAS-95), San Francisco, CA, U.S.A., 1995: 312-319.

[21] Pokahr A, Braubach L, Walczak A, Lamersdorf W. Jadex - Engineering Goal-Oriented Agents. In Bellifemine et al. (Eds): Developing Multi-Agent Systems with JADE, Wiley Publishing, 2007; 254-258.

[22] JADEX User Guide. http://jadex.informatik.uni-hamburg.de/docs/jadex-0.94x/userguide/index.single.html [8 June 2011].

[23] Pokahr A, Braubach L, Lamersdorf W. Jadex: A BDI Reasoning Engine. In Bordini et al. (Eds): Multi-Agent Programming Languages, Platforms and Applications, Springer, 2005; 149-174. DOI: 10.1007/0-387-26350-0_6.

[24] Tool for Agent Oriented Modeling (TAOM4E). http://selab.fbk.eu/taom/ [8 June 2011].

[25] Object Management Group: Model Driven Architecture. http://www.omg.org/mda/ [8 June 2011].

[26] Perini A, Susi A. Automating Model Transformations in Agent-Oriented Modeling. *Lecture Notes in Computer Science* 2006; **3950**: 167-178. DOI: 10.1007/11752660_13.

[27] Eclipse Open Development Platform. http://www.eclipse.org [8 June 2011].

[28] Penserini L, Perini A, Susi A, Mylopoulos J. High variability design for software agents: Extending Tropos. *ACM Transactions on Autonomous and Adaptive Systems* 2007; **2** (4): Article 16, 1-27. DOI: 10.1145/1293731.1293736.

[29] Agent UML. http://www.auml.org/ [8 June 2011].

[30] JADEX BDI Agent System. http://jadex-agents.informatik.uni-hamburg.de/xwiki/bin/view/About/Overview [8 June 2011].

[31] Bellifemine F, Poggi A, Rimassa G. Developing Multi-Agent Systems with a FIPA-compliant Agent Framework. *Software: Practice and Experience* 2001; **31** (2): 103-128. DOI: 10.1002/1097-024X(200102)31:2<103::AID-SPE358>3.0.CO;2-O.

[32] Object Management Group: Object Query Language. http://www.odbms.org/ODMG/OG/ [8 June 2011].

[33] Foundation for Intelligent Physical Agents (FIPA) Communicative Act Library Specification. http://www.fipa.org/specs/fipa00037/ [8 June 2011].