

Model of Negotiation and Decision Support for Goods and Services

Zoltán BALOGH, Michal LACLAVÍK, Ladislav HLUCHÝ
Institute of Informatics, Slovak Academy of Sciences, Bratislava
balogh.ui@savba.sk, laclavik.ui@savba.sk, hluchy.ui@savba.sk
<http://ups.savba.sk/>

Abstract

In the article we propose a model of a centralised support system for negotiation and decision making of software agents. The main entity in our model is represented by a negotiation center. The role of the center is to manage negotiation and support decision-making process of individual agents. Negotiation is based on a Cut Cake algorithm. Presented decision support techniques utilise the theory of economic rationality. Formal definitions of the implemented algorithms are also provided. The model is implemented by using Java IBM Aglets methods and interfaces.

1. Introduction

Agent-based systems technology has become a new paradigm for conceptualizing, designing, and implementing software systems.

Agents are sophisticated computer programs that act autonomously on behalf of their users, across open and distributed environments, to solve a growing number of complex problems.

Increasingly, however, applications require multiple agents that can work together. A Multi-Agent System (MAS) is a loosely coupled network of software agents that interact to solve problems that are beyond the individual capacities or knowledge of each problem solver [11].

Mobile agents are a beneficial technology for the creation of distributed systems. The reasons supporting this statement, are as follows [1]:

- *Agents reduce the network load.* Mobile agents allow to package a conversation and dispatch it to a destination host, where the interaction can take place locally.
- *Agents overcome network latency.* Mobile agents offer a solution to problem of unacceptable latencies in critical real-time

systems, because they can be dispatched from a central controller to act locally and directly execute the controller's directions.

- *Agents encapsulate protocols.* Communication protocols often become a legacy problem. Mobile agents can move to remote hosts to establish channels based on proprietary protocols.
- *Agents execute asynchronously and autonomously.* Mobile devices must often rely on expensive or fragile network connections. Embedding tasks into mobile agents, which can then be dispatched into the network, solves this problem.
- *Agents adopt dynamically,* by the ability to sense their environment and react autonomously to changes.
- *Agents are naturally heterogenous.* Because mobile agents are generally dependent only on their execution environment, they provide optimal conditions for system integration.
- *Agents are robust and fault-tolerant,* because of their ability to react dynamically to unfavorable situations and events.

In today's world we are trying to use our time as best as we can. People value their time much more than before. In many professions and also in everyday life people negotiate and make decisions about what they are going to

buy, where they will go, what they are going to do etc.

Probably we are spending too much time for searching information, than for negotiating and good decision after that. Most of this negotiating is simple or based on certain conditions. Here we see a good use of Intelligent Agents.

2. Research Infrastructures for Intelligent Agent Systems

There are many infrastructures for developing more or less complex distributed multiagent systems. Basically we categorize these architectures as either commercially available products or academic and research projects. We provide a brief survey of selected agent construction tools in this section.

RETSINA

RETSINA is an architecture for developing distributed intelligent software agents that cooperate asynchronously to perform goal-directed information retrieval and information integration in support of a variety of decision making tasks. A collection of RETSINA agents forms an open society of reusable agents that organize and cooperate in response to task requirements. Each RETSINA agent has four reusable modules for communication, planning and scheduling and for monitoring the execution of tasks and requests from other agents.

The RETSINA system research addresses the problem of how to facilitate communication among agents of different types.

Knowbot System Software

The Knowbot software implements a research infrastructure for mobile agents intended for use in widely distributed systems such as the Internet.

The Knowbot software consists of several components. The main component is the Knowbot Operating System (KOS), which creates the platform for running Knowbot programs.

Knowbot programs are experimentally used as "couriers" in an intellectual property rights management system.

The Open Agent Architecture

The Open Agent Architecture (OAA) is a framework for integrating a community of heterogeneous software agents in a distributed environment.

In a distributed *agent* framework, it is possible to conceptualize a dynamic community of agents, where multiple agents contribute by services to the community. When external services or information are required by a given agent, instead of calling a known subroutine or asking a specific agent to perform a task, the agent submits a high-level expression describing the needs and attributes of the request to a specialized *Facilitator agent*. The Facilitator agent will make decisions about which agents are available and capable of handling sub-parts of the request, and will manage all agent interactions required to handle the complex query.

ZEUS

The ZEUS Agent Building Toolkit is an integrated environment for the rapid development of collaborative agent applications. ZEUS is entirely implemented in Java.

The ZEUS toolkit consists of three main functional components:

- *The Agent Component Library*, whose components implement the different aspects of agent functionality,
- *The Agent Building Tools*, which provide an integrated development environment through which the agents are specified and generated,
- *The Visualisation Tools*, the runtime environment that enables applications to be observed and, where necessary, debugged.

JATLite

JATLite (Java Agent Template, Lite) is a package of programs written in the Java language that allow users to quickly create new software that communicate robustly over

the Internet. JATLite facilitates especially construction of agents that send and receive messages using the emerging standard communications language, KQML.

JATLite provides a *template* for building agents that utilize a common high-level language and protocol. The JATLite packaged infrastructure allows agents to be portable, mobile, and to connect and disconnect from the Internet with automatic queuing and buffering of incoming messages.

A primary application of JATLite is to "wrap" existing programs by providing them with a front-end that allows them automatically to communicate with other programs, sending and receiving messages, files, etc.

DECAF

DECAF (Distributed, Environment-Centered Agent Framework) is a toolkit which allows a well-defined software engineering approach to building multi-agent systems. DECAF provides the necessary architectural services of a large-grained intelligent agent: communication, planning, scheduling, execution monitoring, coordination, and eventually learning and self-diagnosis.

DECAF distinguishes itself from many other agent toolkits by shifting the focus away from the underlying components of agent building such as socket creation, message formatting, and the details of agent communication. DECAF provides an environment that allows the basic build block of agent programming to be an agent action.

MASSIVE

Massive kit provides a small, easy, and interactive tool for the fast development of simple applications of multiagent systems. In version 2.0 of the kit, the agents can run on distributed machines. Some features also allow to simulate the notion of mobile agents.

The agents are represented as objects. The user can create a particular multi-agent system application by means of making modifications in the "default agent class", which only offers the basic services. Therefore, the domain and the agents' knowledge specification, as well as

the control flow between them, including the definition of a specific high-level coordination protocol, all depend on the specific definitions in the user programming.

ASDK

The Aglets Software Development Kit is an environment for programming mobile Internet agents in Java.

An aglet (Light weight Agent) is a Java object that can move from one host on the Internet to another. That is, an aglet that executes on one host can suddenly halt execution, dispatch to a remote host, and resume execution there. When the aglet moves, it takes along its program code as well as its state (data). A built-in security mechanism makes it safe for a computer to host untrusted aglets.

3. Negotiation and Decision Support

Negotiation

Negotiation theory was first used in game theory. This negotiation in game theory has one big difference from real life. The game software doesn't have to learn to make good decisions. The game has already all needed information for decision making.

But still in game theory we can find two good ideas:

- In game theory, agents have knowledge of other agent negotiation and decision making algorithm. This can sound strange but this idea can be useful in negotiation.
- Game agents don't have to learn. They have all needed information for negotiating on a certain place and they can use common knowledge instead of its own.

We are describing negotiation strategy similar to game theory below.

What should be supported by Agent Negotiation?

- it should be very similar to human negotiation
- it should be fair
- it should be envy-free

Agent Negotiation is always based on conditions (price > offer, time = time of session etc.). Real life negotiation is not always like this.

Possible negotiation problems and solutions:

- For negotiating strategies is required non-condition decision making. We could possibly use neural networks or some genetic based algorithms. Here we can see the problem of storing and sorting the information. It is very hard for agent to learn as neural networks if it has to carry the information with itself. It is better to create some negotiation-decision centers. Agent then passes only its conditions and states and the negotiation center will provide negotiation for it.
- Fair negotiation is also a problem. In the Negotiation system described bellow is fair negotiation. It is solved by knowing, other agent-negotiating algorithm. In the negotiating center solution this can be solved, because negotiation center increase its knowledge by each negotiating to improve negotiation.
- Envy-free and also fair is Cut Cake algorithm, which is also described bellow.
- Cooperative negotiation – cooperation on the projects, meetings, etc. Goals: - we know here that we can give our (e.g. Timetable data) data to other agent and he will give ours. Agents will agree on some time if possible. If we have some profit function, agents can put profit function together – the best solution is found.
- Non cooperative negotiation: seller – buyer. Here the following problems occur: Seller is selling something between \$10 – \$100. A buyer wants to buy same stuff between \$5 - \$50. If the buyer will say his interval, the seller will sell it for \$50. If the seller will say his interval buyer will buy it for \$10. If we have some profit function, agents can pass profit function to a third agent (negotiation center) and agents get custom information.

Negotiation doesn't have to end-up with rejecting or accepting. The negotiation result can be data for next negotiation, this can be for

example, non-agent but personal negotiation. In this case agents work as a good secretary.

3.1. Formal Description of Negotiation Model.

According to Zeng [10] negotiation process can be modeled by a 10-tuple

$\langle N, M, V, A, H, Q, L, P, C, E \rangle$ where

- A set N (the set of players).
- A set M (the set of issues/dimensions covered in negotiation. For instance, in the supply chain management domain, this set could include product price, product quality, payment method, transportation method etc.)
- A set of vectors $V \equiv \{(D_j)_{j \in M}\}$ (a set of vectors whose elements describe each and every dimension of an agreement under negotiation). A set A composed of all the possible auctions that can be taken by every member of the players set.
 - $A \equiv V \cup \{Accept, Quit\}$
- For each player $I \in N$ a set of possible agreements A_I .
 - For each $I \in N, A_i \subset A$
- A set H of sequences (finite or infinite) that satisfies the following properties:
 - The elements of each sequence are defined over A .
 - The empty sequence Φ is a member of H .
 - If $(a^k)_{k=1, \dots, K} \in H$ and $S < K$ then $(a^k)_{k=1, \dots, S} \in H$.
 - If $(a^k)_{k=1, \dots, K} \in H$ and $a^k \in \{Accept, Quit\}$ then $a^k \notin \{Accept, Quit\}$ when $k = 1, \dots, K-1$.

Each member of H is a history; each component of a history is an action taken by a player. A history $(a^k)_{k=1, \dots, K+1} \in H$. The set of terminal histories is denoted by Z .

- A function Q that associates each nonterminal history ($h \in H/Z$) to a member of N . (Q is called the player function which determines the orderings of agent responses.)
- A set L of relevant information entities. L is introduced to represent the players' knowledge and belief about the following aspects of negotiation:

- The parameters of the environment, which can be changed over time. For example, in supply chain management, global economic and global industry-wide indices such as overall product supply and demand and interest rate, belong to L .
- Beliefs about other players. These beliefs can be approximately decomposed into three categories:
 - Beliefs about the factual aspects of other agents, such as how their payoff functions are structured, how many resources they have, etc.
 - Beliefs about the decision making process of other agents. For example, what would be other players' reservation prices.
 - Beliefs about meta-level issues such as the overall negotiation style of other players. Are they tough or compliant? How would they perceive certain action? What about their risk-taking attitudes? etc.
- For each nonterminal history h and each player $i \in N$, a subjective probability distribution $P_{h,i}$ defined over L . This distribution is a concise representation of the knowledge held by each player in each stage of negotiation.
- For each player $i \in N$, each nonterminal history $H \setminus Z$, and each action $a_i \in A_i$, there is an implementation cost $C_{i,h,a}$. C can be interpreted as communication costs or costs associated with time caused by delaying terminal action (*Accept or Quit*).
- For each player $i \in N$ a preference relation \succsim_i on Z and $P_{h,i}$ for each $h \in Z$ in turn, results in an evaluation function $E_i(Z, P_{z,i})$.

3.2. Negotiation Model of Goods and Services

This model is fair and envy-free. In this model we can see how real life solution would be useful in agent negotiation.

One person divides the cake in two, and the other chooses the first piece. This is fair in that each believes he or she received at least a half,

and envy-free in that neither would wish to trade. If the third person will come each of those two will cut his/her portion into 3 parts and the third person will take one piece from each one.

This protocol succeeds even when participants disagree about the portion's value. [7]

Let's bring this protocol into Agents world. First Agent can use the cake-cutting algorithm when negotiating for resources. For example, two agents sharing CPU time might also wish to complete their computation as soon as possible. One agent might select the sizes of the processor's time slices, while the other would have first choice of the resulting slices. Similar negotiation could occur over bandwidth, relative cache size, locks on databases, storage space in a file system, or task decomposition and distribution.

This algorithm can be applied not only on 2 agents but on N agents as well.

When $n+1$ agents come to negotiate each of n agents will divide its sources into $n+1$ parts. Agent $n+1$ will choose one slice from each agent.

Formal Description of Cut Cake Algorithm

$\langle A, S, F, D \rangle$

A – A set of Agents for negotiation.

S – A set of sources for negotiation. This set is held by one agent at the beginning. S_i is the source held by A_i agent.

F – A set of f_i functions $f_i \in F$, and f_i is cutting function or algorithm of agent A_i .

D – A set of d_i functions $d_i \in D$, and d_i is decision function or algorithm of agent A_i . This function is used for decision making.

3.3. Decision Making and Rationality

According to Doyle [5] judgements of human rationality commonly involve several different conceptions of rationality, including a *logical* conception used to judge thoughts, and an *economic* one used to judge actions or choices.

Intelligence involves both perception and action. One may think of action as simply doing something. But in most cases actions are

not determined by the agent's situation, but instead involve choices to do one thing rather than another. Thus, both thinking and choice are central operations in thinking.

The fundamental issue in the theory of economic rationality is choice among alternatives. Economic rationality means making „good“ choices, where goodness is determined by how well choices accord with the agent's preferences among alternatives.

Preferences

Preference is the fundamental concept of economic rationality.

We write $A \succ B$ to mean that agent prefers B to A , and $A \sim B$ to mean that the agent is indifferent between the two alternatives. We also write $A \not\succeq B$ to mean that $A \succ B$ does not hold and $A \not\sim B$ that either $A \sim B$ or $A \succ B$. The collection of all these comparisons constitutes the agent's set of preferences. These preferences may change over time.

Rational agents choose maximally preferred alternatives. If $\alpha = \{A_1, A_2, \dots, A_n\}$ is the set of alternatives, then A_i is a rational choice from among these alternatives just in case $A_i \not\succeq A_j$ for every $A_j \in \alpha$. There may be several rational choices, or none at all if the set of preferences is inconsistent or if the set of alternatives is empty or infinite.

The theory does not require that a rational agent explicitly calculates or computes the maximality of its choices, only that the agent chooses alternatives that are in fact maximal according to its preferences.

The theory requires, as a minimum basis for rationality, that strict preference is a strict partial order, indifference is an equivalence relation (transitive and asymmetric), and any two alternatives are either indifferent or one is preferred to the other, but not both at the same time. These separate requirements on preference and indifference may be put formally, for all alternatives A, B and C :

1. Either $A \succ B$ or $B \succ A$, (completeness)

2. If $A \succ B$, then $A \not\succeq B$, (consistency)

3. If $A \succ B$ and $B \succ C$, then $A \succ C$. (transitivity)

The rationality constraints imply that we may represent the set of preferences by means of a numerical utility function u which ranks the alternatives according to degrees of desirability, so that $u(A) < u(B)$ whenever $A \succ B$ and $u(A) = u(B)$ whenever $A \sim B$. By working with utility functions instead of sets of preferences, we may speak of rational choice as choosing so as to maximize utility. The same set of preferences may be presented by many utility functions, as any strictly increasing transformation of a utility function will provide the same choices under maximalization.

The distinction between the cost or values of something and its utility or disutility is one of the great strengths of the theory of economic rationality. The cost of some alternative says nothing about the value or benefits received from it, and neither cost nor benefit need be identical with the utility of the alternative if the risk posed by the action diminishes (or increases) its attractiveness to the agent. The utility of an action is usually some function of the cost, benefit, risk and other properties of the action.

Decision Theory

Compared with the basic theory, decision theory adds probability measures p_A which indicate the likelihood of each possible outcome for each alternative A . Decision theory supposes that the agent does not know the actual situation, but does have beliefs or expectations about the consequences of choice in different states.

Decision theory also strengthens the notion of utility from an *ordinal* utility function u to a *cardinal* utility function U , which inputs different values to each possible outcome. Ordinal utility functions use numeric values simply as ways of ranking the alternatives in a linear order. Amounts of cardinal utility can be added and subtracted to produce other amounts of utility. This makes it possible to combine the utilities foreseen in different

possible outcomes of A into the expected utility $U(A)$, defined to be the utility of all possible outcomes weighted by their probability of occurrence, or formally,

$$U(A) = \sum_S^{def} p_A(S)U(S),$$

where the sum ranges over all possible situations or states of nature under discussion. The decision-theoretic definition of preference is

$$A \succeq B \text{ if and only if } \hat{U}(A) \leq \hat{U}(B).$$

Like the theory of preference, the assumptions of decision theory can be also formulated qualitatively [5].

4. Implementation of Goods and Services Algorithm

Implementation it is done in Java Aglet Software. We have chosen this agent interface because Java is getting more powerful, and used Internet language. Java is supporting secure migrating of classes but programming of Mobile Agents needs some additional useful properties. All this is covered in IBM Aglets which are still under development. There is no sense in creating other Mobile Agent Interface in Java. Therefore we decided use Aglets. Java programming language is under development also and lot of useful classes can be found on the internet. IBM also created JKQML – implementation of KQML in Java. Java is also supporting SQL access to many SQL servers so it is possible to plug agents to real world applications when they are created in Java [1, 5].

KQML

KQML is a language for the communication between software agents.

KQML offers a variety of message types (performatives) that express an attitude regarding the content of the exchange. Performatives can also assist agents in finding other agents that can process their requests.

Performatives examined in the KQML specification [8] are organized in three categories:

- *discourse performatives*, which can be considered as close as possible to speech acts in the linguistic sense,
- *intervention and mechanics of conversation performatives*, whose role is to intervene in a normal course of conversation and
- *networking and facilitation performatives*, primarily used to allow agents to find other agents that can process their queries.

A notion of domains is one of the fundamental concept of the KQML.

In each domain of KQML-speaking agents there is at least one agent with a special status called facilitator. Agents advertise to their facilitator, i.e., they send advertise messages to their facilitators, thus announcing the messages that they are committed to accepting and properly processing.

Agents can access agents in other domains either through their facilitator, or directly.

Facilitators may request the services of other facilitators in the same way that regular agents may request the services of their facilitator. Facilitators do not advertise, not even to other facilitators.

Here is an example of a typical KQML expression, where `bt-neg-center` agent replies to `customer-agent-2` agent using `standard-prolog` language.

```
(tell
  :content „cost (bt, service-4,
             $5677) “
  :language standard-prolog
  :ontology bt-services-domain
  :in-reply-to service-4
  :receiver customer-agent-2
  :sender bt-neg-center)
```

We implemented the negotiation model of goods and services (cut cake algorithm). We point out some features here:

Negotiation centre – because of learning, information storage and coordination problems, it is not good for each agent to learn

and store all information itself. There is also problem into finding agent for negotiation (coordination) [4]. By providing negotiation centre these problems are solved. Negotiation center is easier to access. It has better information about services and decisions because all negotiation between agents is via negotiation center. It also stores information on one place.

If new learning, data access or other strategies will be added, we don't have to modify all agents but only the negotiation center.

Cut cake algorithm – this algorithm can be used for many situations. Negotiation by this algorithm is really fast. Agent sends an offer and negotiation center will send the offer to agents which can possibly deal with it. The result is going back to the first agent, which can decide which „cake“ it will take. Here is a simple example what can it be used for.

Agent wants to buy a computer journal for one year at price not higher than \$100 per year.

Negotiation center can pass to agent to choose accept journal which costs \$200 per year but for half a year it will be for \$100 or a journal for \$100 for 2 years etc. Agent will pass this information to a person via email or to some interface or will decide which choice is the best or reject all offers.

As negotiation language we using KQML. This solution is also suitable for future improving of our example to a real commercial negotiation system. KQML is implemented in many other agent developing systems such as ARA, TCL and also Aglets [3]. This way agents implemented in other languages can be used to negotiate with the center.

KQML is implementing email KQML support as well. This means that simple email messages containing KQML request sent to KQML agent are behaving as agent. This can be used very well on unix machines.

Incorporating Decision Support Algorithms

We see a need for implementing a rational decision making algorithm in individual agents of our experimental system as a central part of the „cut cake“ algorithm.

Negotiation center uses the decision algorithms to rationally apportion goods and services into parts with equal utilities. In order to enable the system to compute utilities, negotiation center needs to know the utility function of particular goods and services. Individual utilities are functions of elements such as price, amount, time, etc.

On the other hand, client agents need to make the best choice among alternatives offered by the negotiation center. We suppose that each client agent aims at maximizing the total agent's utility while choices must accord with the agent's preferences.

Implemented classes

NegCenter class

- this class represents the negotiation center. It is a subclass of the main Aglet class. Agents can access it with KQML requests for negotiating. Class stores information about services into MySQL database via an JDBC interface. The following information is stored: name, category, description, minimum quantity, maximum quantity, value (price). This information is stored when agent sends a bid to negotiation center. It is used when agent sends offer to center. It is also sorted by prices and possibilities and passed to agent.

AgentNeg class

- this is a subclass of Aglets and JKQML. It is supporting KQML requests and NegCenter specific communication functions.

AgentBidder class

- this is a subclass of AgentNeg class. It can be used as agent which is bidding some services (seller).

AgentOffer class

- this is a subclass of AgentNeg class. It can be used as agent which is offering some services (buyer).

3. Conclusion and Future Work

This article describes existing multi-agent systems and models. Also describes some negotiation models and their improvements. Implementation of negotiation models in an agent development system is also given.

There is lot of future work what can be done in this area. This will be also future work of our research.

The learning of negotiation model can be improved, and other learning methods such as neural network etc. can be used as well The negotiation model is scalable for whatever learning method so it is posible to build on described model.

The implemantation of this negotiation is simple and abstract. It can be made more powerful when database support and internet communication support will be added. This will make our experiment grow to Agent system what can be easily used for many applications with a little programming only.

References

[1] Bigus, J., Bigus, J.: Constructing Intelligent Agents with Java. Willey Computer Publishing, 1997, Canada

[2] Chuang, T., Yadav, S., B.: An Agent-Based Architecture Of An Adaptive Decision Support System, 1997.

[3] Cockayne, W., R., Zyda, M.: Mobile Agents, Manning Publication co., 1997. USA

[4] Cremonini, M., Omicini A., Zamboneli, F.: Ruling Agent Motion in Structured Enviroments, 2000. HPCN Europe

[5] DOYLE, J.: Rationality and its Roles in Reasoning. In Computational Intelligence, Vol. 8, No. 2 (May 1992), pp. 376-409.

[6] Genesereth, M., R., Ketchpel, S. P.: Software Agents.

[7] Huhns, M., N., Malhotra, A., K.: Negotiating for Goods and Services, IEEE Internet Computing, July-August 1999.

[8] Labrou, Y., Finin, T.: A Proposal for a new KQML Specification, TR CS-97-03, available through the Stanford University Computer Science Department, 1997.

[9] Lange, D., B., Oshima, M.: Programming and Deploying Java Mobile Agents with Aglets. Addison-Wesley, 1998. Canada

[10] Zeng, D., Sycara, K.: How Can Agent Learn to Negotiate? In: Intelligent agents III, ECAI '96 Workshop, 1996, pp.233-244

[11] Intelligent Software Agents.
[<http://www.cs.cmu.edu/~softagents/>]