Università degli Studi di Genova
Facoltà di Scienze Matematiche Fisiche e Naturali
Corso di Laurea in Informatica

Anno Accademico 2004/2005

Tesi di Laurea

Aste elettroniche in ambiente multi-agente

Candidato
Davide Roggero

Relatori

Prof. Fioravante Patrone

Dott. Viviana Mascardi

Correlatore

Dott. Giorgio Delzanno

A mia nonna,
che, dovunque sia,
e' con me.

Ringraziamenti

Inizio con i ringraziamenti a coloro che mi hanno aiutato nella realizzazione di questo lavoro.
Ringrazio il prof. Fioravante Patrone e la dott. Viviana Mascardi per l'aiuto fornitomi, per la disponibilità e per avermi portato (in certi momenti anche di peso!) alla fine del percorso.
Un grazie anche al mio correlatore, dottor Giorgio Delzanno.
Un ultimo ringraziamento tecnico, alla dottoressa Ivana Gungui, senza la quale il mio lavoro di tesi (e il mio capitolo di introduzione a DCaseLP) sarebbero stati più faticosi.
Ora passiamo ai ringraziamenti personali:

- Ai miei genitori, perché sono i miei genitori e, nella lite e nella pace, non li scambierei con altri;
- al mio amore, quella Foglia che mi é caduta fra le mani, che mi sostiene e mi stimola sempre ad andare avanti;
- alla mia nonna, che é scomparsa alla vista di tutti in un inverno freddo, ma non é mai scomparsa dal mio cuore;
- ai miei amici dei tempi del liceo, Andrea, Carlo e Luca, perché sono quasi dei fratelli per me;
- a Chiara, Sara e Marta perché sono pezzi importanti della mia vita;
- ai miei amici di vecchia data, Alessandro, Cristiano, Massi, Chiarina, Gloria e Sabrina, per la loro compagnia e le serate passate insieme;
- a Gabriele, Daniele, Flavia, Marcolino, Francesca e a tutti gli amici con cui ho passato tanto (TROPPO!) tempo qui all'università;
- a tutti gli amici del teatro, con cui ho condiviso momenti divertenti;
- a tutta Calvari, perché é un covo di malati di solidarietà;

e a tutti gli altri amici, sono troppi da elencare e deve ancora stampare questa tesi!

Contents

Introduction

Information and Communication Technology (ICT) is currently considered as one of the forces that can deeply influence and transform human society. Many people agree on the important role played by ICT in productive growth and international competitiveness, thanks to reduction of transaction costs, support to efficient management, and exchange of and access to a greater number of information. This happened especially for commerce, changing radically how enterprises and economies work. For example, large on-line selling enterprises use the Internet strategically to improve service quality, process speed and for cost savings, whereas small enterprises use e-commerce primarily to increase their customer base and make themselves known.
This trend continues and will have even deeper impact on many commercial areas: Agent-Mediated Electronic Commerce (AMEC) can offer potential answers to open challenges like business process outsourcing, marketing of agricultural exports and online dispute resolution.
C.Sierra, in his article 'Agent-Mediated Electronic Commerce' [Sie04], asserts that electronic commerce can be described by the following equation:

$$eCommerce = organization + mechanism + trust$$

Organization. Problems of co-ordination and co-operation have been a challenge for human society for centuries. To solve these problems, norms and rules have been established, and organizations and institutions have been created to back up those norms and enforce them. The success of these social abstractions in providing an answer to the co-ordination demands of citizens has served as inspiration for multi-agent systems designers facing similar complex problems. In the context of commerce and trade, there are many organizations and institutions that provide support for human interactions; hence, it is not strange that electronic commerce developers have had a special interest in the use of these metaphors. In a sense, social order is the multi-agent system equivalent of robustness in classical software engineering.

The distributed nature of multi-agent systems and the flexible interactions among its agents determines the complexity of its design. Furthermore, this complexity increases notably as we consider *open* systems; that is, systems in which the components are not known in advance and/or can change over time. In these systems, agents cannot be assumed to be cooperative and working for the common good of the overall multi-agent system. Usually, assumptions of agents being selfish (utility maximizers, in economic terms) are made by the system designers. The expansion and massive usage of networks makes openness a necessary requisite in the development of multi-agent systems, because of dynamic nature in which services and devices appear and disappear continuously. Dynamism becomes commonplace.

Organizational approaches seem to be the solution to this challenge. They propose to analyze the system from a global viewpoint, determining roles and groups of agents and their interactions and relationships, without looking into their inner characteristics. In recent years, the interest in organizational approach has grown, particularly within groups working on electronic commerce applications: Gaia [WJK00], MadKit [FG98], Electronic Institutions [Rod01], and Tropos [GMP02] are systems that uses this methodological approach.

Mechanism. Commerce is all about interaction between buyers and sellers at all stages: finding, purchasing, and delivering. In order to support interaction, autonomous agents are increasingly being used in a wide range of industrial and commercial domains.

Autonomous agents have a high degree of self-determination they decide for themselves what, when and under what conditions their actions should be performed. In most cases, such agents need to interact with other agents to achieve their objectives (either because they do not have sufficient capabilities or resources to complete their problem-solving alone, or because there are interdependencies between the agents). The objectives of these interactions are to make other agents undertake, modify or cancel a particular course of action (e.g. perform a particular service) or come to an agreement on a common course of action.

Mechanism design is precisely concerned with fixing the rules governing the interaction among agents in such a way that certain properties (such as stability, or equilibrium) can be guaranteed. The definition of the rules of the game determines how the interaction will take place and, introducing constraints on the complete autonomy of agents, tries to induce a given (rational) behavior in them, if possible by selecting dominant strategies.

Trust. In most real cases, organizations and protocols cannot completely guarantee that agents will behave as expected, or as agreed upon in a contract. Human societies and, naturally agent societies as well, have to face risks in interaction. And trust between agents has proved to be a good way to reduce risks.

Distributed systems formed by thousands or millions of agents necessarily require new mechanisms to deal with security. This is especially important in electronic commerce environments where transactions involve a significant amount of money. Traditional methods based on Access Control Lists (ACL), or on Role-based Access Control, stop working when the individuals may not be known ahead of time, as happens in open multi-agent systems. Different approaches have been proposed to overcome this situation: using chains of trust, rights, and delegation.
The uncertainty relating to the behavior of an agent in a society can be perceived as a potential source or risk in a commercial transaction. This is why it is essential to find ways of removing this uncertainty if electronic commerce is going to be realized in open environments. Trust and reputation measures are the inspiring social mechanisms that researchers in the field of electronic commerce are looking into in order to increase the number of transactions. Here, we understand trust as the positive expectation that a partner will act cooperatively in situations in which defection would prove more profitable to itself.
In recent years, there has been an explosion of reputation models that try to build trust in electronic commerce transactions. The models used by eBay, Amazon and OnSale Exchange, although rather simple, are good examples. All of them use some sort of average of user opinions. But other more complex reputation systems, based on different reputation measures and using fuzzy sets, have been proposed: for example, REGRET [Sab03] and CREDIT [RSGJ03].

The aim of the thesis

Looking at the Sierra's equation, we focused on the mechanism aspect of electronic commerce, developing a *library of customizable agents for simulating auction mechanisms*. The goal is to provide tools for the strategical analysis of realistic auctions and for comparing theoretical results with empirical tests. The need for a prototyping method and a set of tools and languages to support the analysis and realization of auction mechanism prototypes brought us to choose the DCaseLP environment.
DCaseLP [Mig02, AMMM02] means ***D****istributed CaseLP*, where **CaseLP** [MMZ99, BDM$^+$99] is the acronym for ***C****omplex* ***A****pplication* ***S****pecification* ***E****nvironment based on* ***L****ogic* ***P****rogramming*

DCaseLP (Distributed CaseLP) [Mig02, AMMM02], is a rapid prototyping software environment that supports the development of MASs and, as its predecessor CaseLP [MMZ99, BDM$^+$99], has been designed and developed by the Logic Programming Group at the Department of Computer Science of the University of Genova in Italy.

DCaseLP aims at providing the developer of a MAS with an AOSE methodology and a software environment to be used during the requirements analysis, the design and the development of a working prototype. A fundamental goal is to support the development of MASs consisting of multilingual agents. More precisely, more than one language must be available not only to specify the agents belonging to the system, but also to define their architecture, behavior and state, allowing both existence and communication in the environment of agents created using such different languages.
For this thesis, we decided to use **AUML** [FIP] as the specification language for the analysis of agents interaction, while we used Pascal pseudo-code to design the internal behavior of the agents.
Logic programming language **tuProlog** [DOR01] was chosen for the implementation of the prototypes because its characteristics, like knowledge representation and support to agent autonomous reasoning, seemed the most suitable to accomplish our tasks.

Structure of the thesis

This thesis is made of two parts, the first three chapters are devoted to introducing the theoretical background of this thesis, while the second part is devoted to describe the technical aspects.
In chapter 1 we introduce the basics of Decision theory, that is the mathematical theory that studies models with one decision-maker. We introduce the notions of *preference*, *utility functions* and *risk*.
Chapter 2 illustrates some models and results of Game Theory, that is the mathematical theory of conflict and cooperation situations. In the models proposed by Game Theory each participant, called *player*, has to decide *rationally* what is the *strategy* to play (what course of actions to undertake), taking in account both his/her preferences over the correspondent outcomes of the game and the other player's strategies. We introduce also the fundamental notion of *Nash equilibrium*, a solution of the game by which all the players are satisfied.
In Chapter 3 we show the results obtained by applying the models and theorems of Game Theory to the analysis of auctions. This set of mathematical results and models goes under the name of Auction Theory and includes the famous Revenue Equivalence Theorem (RET).
Chapter 4 describes DCaseLP and **JADE**(Java Agent DEvelopment Framework), a software development framework aimed at developing multi-agent systems and applications conforming to **FIPA** (Foundation for Intelligent Physical Agents) standards for intelligent agents. It includes two main products: a FIPA-compliant agent

platform and a package to develop Java agents. It also offers a set of graphical tools to support the debugging and deployment phases.
Chapter 5 deals with analysis and design of auction mechanisms. We analyze four different auction mechanisms, producing an Interaction Protocol for each mechanism. Then, in the design phase, we show the pseudo-code of each agent implemented.
In Chapter 6, we describe the customizable characteristics of each mechanisms implemented, then we test the prototypes implemented under the hypothesis of the RET and we verify the correspondence of empirical results with theoretical ones.

Chapter 1

Decision Theory

Game theory deals with situations in which more than one decision makers are involved. Before considering game theory, it is useful to have a look at the basics of "decision theory", that is, the case of a single decision maker. There are three fundamental cases that are considered:

- decision making under certainty
- decision making under risk
- decision making under uncertainty

We shall first discuss decision under certainty; then, we shall discuss decision under risk, while we shall almost entirely avoid the treatment of decision under uncertainty.

1.1 Decision under certainty

The standard model about decision under certainty is given by the couple $(X, \succeq)$, where X is a non empty set and $\succeq$ is a relation on X. There are usually some assumptions on $(X, \succeq)$, that are collected in the following Table 1.1. On the left is the name of the property, on the right is written the intended meaning (interpretation) of the property.

X is a set	the set of available alternatives
$\succeq$, a relation on X	the preferences of the decision maker
$\succeq$ is reflexive	no serious meaning
$\succeq$ is transitive	coherence condition: essential assumption about the rationality of the decision maker
$\succeq$ is total	the decision maker can always express his preferences w.r.t. any couple of alternatives

Table 1.1

Let's fix the mathematical terminology:

Definition 1 *A relation $\succeq$ on X is said to be a preorder if it is reflexive and transitive. It is said to be a* total preorder *if it is a preorder and is also total. We say that a relation $\succeq$ defined on X is:*
reflexive, if: $\forall x \in X, x \succeq x$
transitive, if: $\forall x, y, z \in X, \quad x \succeq y$ *and* $y \succeq z$ *implies* $x \succeq z$
total, if: $\forall x, y \in X, \quad x \succeq y$ *or* $y \succeq x$

Some remarks about the interpretation of these properties. In this thesis we use a weak approach to preferences, but we could have introduced a strict relation of preference $\succ$ and obtain similar results. The choice of using the weak or the strict approach is in general just a matter of taste. The strict approach is adopted in books by Fishburn[Fis79] and Kreps [Kre88], while the weak approach can be seen in Myerson [Mye91].
The key condition to model conflict situation is transitivity. It is the basic ingredient for rationality as is usually intended by economists.
In fact, the core of game theory is the analysis of interactions among *rational* decision makers. So, transitivity will be assumed in what we shall do. Clearly, there is room for models about decision makers which do not satisfy the transitivity assumption, but the "core model" assumes that. Many example can be provided (e.g. "money pump" in Binmore[Bin92]) to show that the absence of transitivity can lead to weird results.

About the interpretation of the "total" property, it does not exclude indifference between alternatives, but excludes that the decision maker is unable to compare a couple of choices. This is a strong assumption, and in some cases is patently violated. However in many instances it is not a too severe restriction, and since it allows to work in a much easier way, it is customary to assume it. Clearly, whenever modelling a decision-making situation, this assumption (as any other one) has to be tested for realism (or reasonableness).
To conclude, the core model for decision making under certainty is formally given by a couple $(X, \succeq)$, where X is a non empty set and $\succeq$ is a total preorder on X.

Some formal developments of the model
It is known that, given $\succeq$ on X, one can define the "dual" relation $\preceq$. The definition is the following:

$$x \preceq y \quad :\Leftrightarrow \quad y \succeq x$$

Given $\succeq$, a couple of more interesting relations can be defined (actually, both of them have been already used in the informal discussion of the basic model):

$$x \sim y \quad :\Leftrightarrow \quad x \succeq y \text{ and } y \succeq x$$

$$x \succ y \quad :\Leftrightarrow \quad x \succeq y \text{ and } \neg(y \succeq x)$$

The symbol $\neg$ is the usual symbol for the negation of a relation.
The first relation above will be referred to as the "indifference" relation, while the second is the "strict preference" relation (this is the terminology used in the standard interpretation of the model). Another useful derived relation is $x \prec y$, defined as $y \succ x$.
Given a total preorder $\succeq$, we can say that $\preceq$ is also a total preorder and that its dual is the original $\succeq$. For $\sim$, it is easy to check that it is an equivalence relation (reflexive, symmetric and transitive). Reflexivity and transitivity for $\sim$ are consequence of reflexivity and transitivity of $\succeq$, while symmetry is a consequence of the structural symmetry of its definition.
The crucial point is anyway the relation $\succ$. These are its essential properties:

Theorem 1 *Given $(X, \succeq)$, with $\succeq$ total preorder, the relation $\succ$, defined as above, is:*

- *asymmetric* ($\nexists x, y \in X$ *s.t.* $x \succ y$ *and* $y \succ x$)
- *negatively transitive* ($\forall x, y, z \in X : (x \succ y \Rightarrow (x \succ z$ *or* $z \succ y))$)

The $\succ$ relation has other additional properties, but those stated in Theorem 1 are crucial, as can be seen by the following:

Theorem 2 *Given $(X, \succ)$, with $\succ$ asymmetric and negatively transitive, then the relation $\succeq$, defined as follows:*

$$x \succeq y :\Leftrightarrow \neg(y \succ x)$$

is a total preorder.

Proposition 1 *Given $(X, \succeq)$, define $\succ$, and then from $\succ$ define "$\succeq$" as done in the theorem above. Then, the last relation coincides with the original $\succeq$. And similarly if one starts with $\succ$.*

Utility functions

Decision problems can be reduced to finding the outcome x_b in a subset X_B of X that the decision-maker most prefers. A problem of this kind looks easy when stated in this abstract way but it can be hard to solve if the elements of X are complicated and the preference relation $\succeq$ is difficult to describe.
A mathematical device, strictly related to preferences, can be introduced to simplify the situation: utility functions.

Definition 2 *Given $(X, \succeq)$, where $\succeq$ is a total preorder, a function $f : X \to \mathbb{R}$ s.t.:*

$$x \succeq y \quad \Leftrightarrow \quad f(x) \geq f(x)$$

is said to be a utility function representing $\succeq$.

Thanks to this definition, a decision problem can be stated in this way: find a value x_b in a subset X_B of X such that $f(x_b) = \max_{x \in X} f(x)$.
It is important to underline the fact that utility functions have been introduced because are useful to solve problems mathematically, but we do not claim that people really have utility generators inside their head: we can say that rational players will behave as though their aim is to maximize some utility function. On the other hand, it is reasonble to think that real rational players have preferences over a set of outcomes. If X is a finite set, it is easy to provide an explicit, algorithmic construction for a utility function. Assume that $X = \{x_1, \ldots, x_n\}$. Start from x_1 and define $f(x_1) = 0$. Then, take x_2. If $x_2 \sim x_1$, then define $f(x_2) = 0$. If $x_2 \succ x_1$, then define $f(x_2) = 1$, and if $x_1 \succ x_2$, then define $f(x_2) = -1$. Then look at x_3. If x_3 is strictly preferred to all of the preceding elements, define $f(x_3) = 2$, and similarly if it is the worst one. In case of indifference with a previous one, the value assigned to $f(x_3)$ will coincide with the value assigned to the indifferent element(s). If x_3 lies strictly between the two preceding elements, define $f(x_3)$ as the mean value

of $f(x_1)$ and $f(x_2)$. The construction proceeds in the same way for the remaining elements of X. Just remark that, given x_k, if there is one element strictly preferred to x_k and one strictly worse, one has to locate the elements of X which are closer (in the sense of the preorder) to x_k, and then to define $f(x)$ as the mean value of the values assigned to these elements closest to x_k.
In the case in which X is countable, just take $X = \{x_1, \ldots, x_n, \ldots\}$ and use the same procedure. There are enough real numbers to comply with the required procedures of taking mean values.
The answer is much less obvious in the general case. It is easy to predict that continuity properties will have some role, since otherwise using uncountable sets one can easily provide extremely wild examples.
In the general case, there is no guarantee that we can find a utility function that represents a total preorder. For example, the lexicographic ordering cannot be represented by any utility function.
However, there are some conditions which guarantee the representability of a total preorder, and they can be easily adapted to deal with preorders.

Theorem 3 *Let be given $(X, \succeq)$, with $\succeq$ total order on X. There is $f : X \to \mathbb{R}$ representing $\succeq$ if and only if there exists $W \subseteq X$, countable and "order dense" in X*

A reference for a proof of this theorem is Fishburn[Fis79]. The meaning of "order dense" is given by the following

Definition 3 *Given $(X, \succeq)$, with $\succeq$ total order on X, $W \subseteq X$ is said to be "order dense" in X if*

$$\forall x, y \in X \backslash W \text{ s.t. } x \succ y, \exists z \in W \text{ s.t. } x \succ z \succ y$$

A famous example of an "order dense" subset is given by $\mathbb{Q}$, seen as a subset of $\mathbb{R}$, with the usual order on $\mathbb{R}$.
Just few words on the fact that the theorem deals with orders instead of preorders. Given $\succeq$, total preorder, we can introduce $\sim$ which is, as already noticed, an equivalence relation. Hence, we can consider the quotient space $X/\sim$, that is, the set of equivalence classes w.r.t. this equivalence relation.
Then, we can define $[\succeq]$ on $X/\sim$ in the following way:

$$[x] \succeq [y] :\Leftrightarrow x \succeq y$$

Clearly, one has to check that the definition does not depend on the representative elements chosen in the equivalence classes. But this is straightforward. The final

result will be that $[\succeq]$ is a *total order* on $X/\sim$. With the use of the quotient space we have obtained that the indifferent elements collapse into a single one (the equivalence class).

Uniqueness of the utility function
Given $(X, \succeq)$ and assuming that there is a utility function $f : X \to \mathbb{R}$ which represents it, it is easy to verify that $2f$, $f + 1$, represent the same preorder. Moreover, given any $\phi : \mathbb{R} \to \mathbb{R}$ strictly increasing, $\phi \circ f$ also represents $\succeq$. It can be proved the following:

Theorem 4 *Let be given $(X, \succeq)$, with $\succeq$ total preorder on X. Then $f, g : X \to \mathbb{R}$ represent $\succeq$ if and only there is $\phi : f(X) \to g(X)$, strictly increasing, onto and s.t. $g = \phi \circ f$.*

1.2 Decision under risk

Let us start with an example. A gambler plays at the roulette. That is, he bets 1000 euro on number 23. He does not know the consequence of this action, but he knows that there is a chance over 37 that number 23 will come out, while there are 36 chance over 37 that a different number will come out (the "game" is slightly unfavourable to the player, due to the presence of the 0 in the roulette). Thus, there is probability of $1/37$ that he will gain 35000 euro, and probability of $36/37$ that he will lose 1000 euro.
This is the typical situation of *decision under risk*. These are the common settings: for each action of the player there is a set of possible results and a probability distribution over these results; the player has to choose between actions, thus he has to compare between probability distributions to decide what is the action to choose: it means that he must have preferences over probability distribution (that decision theorists call *lotteries*).
We have a set X, and we are interested into preferences $\succeq$ on $\Delta(X)$ (where $\Delta(X)$ is the set of probability distribution on X).
Notice that, if X is a finite set, there is no problem in considering the set $\Delta(X)$ of all probabilities on X. But, if X is some set like $[0, 1]$, there are some details that must be fixed. What is a probability on $[0, 1]$? It is a measure with total mass 1, but one has to specify the σ-algebra on which the measure is defined (in general, you cannot assume that you can attach a probability to any subset of $[0, 1]$). So, in the general case, there are some technical complications. To avoid them, we shall restrict the set of probability distributions that we consider. We shall look only at *simple* probability distributions on X.

Definition 4 *Given a set X, a simple probability on X is a function $p : X \to [0, 1]$ s.t. $p(x) \neq 0$ only for a* finite *subset of X, that will be called the support of p, and denoted by* spt(p).

So, the setting in which we will work is the following.
We have a set X, $X \neq \emptyset$. The intended meaning of the elements of X is that they are the "final consequences". Then, we consider P, the set of simple probability on X (we shall also refer to the members of P as to lotteries). That is, $P = \{p : X \to [0, 1]$ s.t. spt(p) is finite, and s.t. $\sum_{x \in X} p(x) = 1\}$.
Notice that $\sum_{x \in X} p(x) = \sum_{x \in \text{spt}(p)} p(x) = 1$, that is, it is essentially a finite sum.
A simpler setting could be assuming that the set X is itself a finite set. Doing this, however, it would be impossible to work with some kind of problems that we have in mind (essentially, probability distributions on monetary consequences).
Regarding preferences, the setting is similar to that of decision making under certainty, but with an essential difference: now preferences are on P whereas, in the previous case, preferences were on X itself.
The set P is not "any" set (as it was in the case of decision making under certainty), but it has a special mathematical structure which shall be taken into account in the assumptions we shall make.
It is useful to remark that the simplest examples of lotteries (i.e., elements of P) are given by concentrated measures, that is: $p = \delta_{\{x\}}$ (the Dirac's delta, concentrated on x, which assigns probability 1 to x and 0 to any other element of X). It is natural to identify X with this set of probability distributions, so that X can be seen as a subset of P. Although it is not equivalent to say that you can obtain x with certainty or that you can get it with probability 1, for our purposes we shall not make such a distinction.

Given a set X, P and a relation $\succeq$ on P, we shall assume

Assumption 1 *$\succeq$ is a total preorder on P*

From an interpretational point of view, this assumption is *stronger* than the analogous one made for decision making under certainty. This is because we are dealing with more complicated objects. For example, our decision maker now has to be able to compare any couple of (simple) probability distributions on the consequences of his actions, which is a much stronger requirement than asking to be able to compare any couple of consequences.
The following assumptions are entirely new w.r.t. the case of decision making under certainty.

Assumption 2 *(archimedean condition)* $\forall p, q, r \in P$ *s.t.* $p \succ q \succ r, \exists \alpha, \beta \in]0, 1[$ *s.t.* $\alpha p + (1 - \alpha)r \succ q \succ \beta p + (1 - \beta)r$

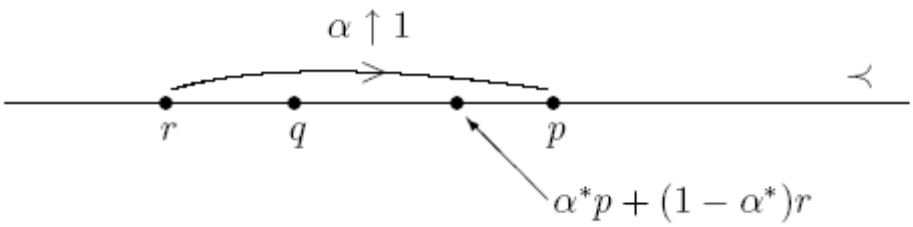

Figure 1.1: The archimedean property

Assumption 3 *(independence)* $\forall p, q, r \in P$ *s.t.* $p \succ q, \forall \gamma \in]0, 1[, \gamma p + (1-\gamma) r \succ \gamma q + (1-\gamma) r$

A notation like $\alpha p + (1-\alpha) r$ is a representation of a convex combination of elements of P, operation which is meaningful on simple probabilities. A similar kind of operation is not possible, in general, in the context we used in the case of decision making under certainty, this means that assumptions 2 and 3 in that context are, generally speaking, meaningless.

Coming back to the interpretation of assumptions 2 and 3, we shall use extensively these concentrated measures. Thus, let us take $p = \delta_{\{x\}}$ and $q = \delta_{\{y\}}$. In assumption 3 there is the hypothesis that the decision maker strictly prefers p to q (which means essentially that the player prefers x to y). Now, let us consider another element of P, for example $r = \delta_{\{z\}}$. The idea of assumption 3 is that a decision maker who prefers p to q is *obliged* to prefer, for example, to get x with probability $1/3$ and z with probability $2/3$ to getting y with probability $1/3$ and z with probability $2/3$.

To understand such a restriction, one can imagine that has to choose between a device giving x with probability $1/3$ and z with probability $2/3$ and a device giving y with probability $1/3$ and z with probability $2/3$. Now, there is a probability of $2/3$ that the final outcome will be the same, i.e. z, and there is a probability $1/3$ that the final outcome will be x or y. Since the decision maker strictly prefers x to y, he should also strictly prefer the first device to the second one. To put things clearer, if one strictly prefers to receive 1 euro instead of receiving nothing, then one should strictly prefer to receive 1 euro with probability $\alpha > 0$ to receiving nothing.

To illustrate assumption 2, consider three lotteries $p = \delta_{\{x\}}$, $q = \delta_{\{y\}}$, $r = \delta_{\{z\}}$, which can be represented on a line as in Figure 1.1 below:

Then, one considers $\alpha p + (1-\alpha) r$, which can be written also as: $r + \alpha(p - r)$. Consider "moving" α from 0 to 1: it happens (in this representation) that this lottery moves from r to p. So, *in this representation*, it is natural to assume that there is

some α^* s.t. $r + \alpha^*(p - r)$ lies "to the right" of q. This is precisely the meaning of the archimedean property. It guarantees that a result like this, suggested by the picture made, really holds. This fact could not happen if it were the case that some alternative, some lottery, is not "comparable" with other alternatives. That is, it is "infinitely worse" (or "better") then another one. In such a case, maybe it is not possible to make a convex combination of lotteries capable of compensating the occurrence of "hell", that is an alternative which is infinitely worse compared with another one. In the picture, r could be imagined to be located at "$-\infty$". Then there is no convex combination of r and p such that the decision maker could consider it better than q.
We are now able to state the representation theorem.

Theorem 5 *Let be given P, set of simple probability distributions on some set $X \neq \emptyset$, and a relation $\succeq$ on P. Then, $\succeq$ satisfies assumptions 1,2 and 3 if and only if there exists $u : X \to \mathbb{R}$ s.t.:*

$$(*) \qquad p \succeq q \Leftrightarrow \sum_{x \in \text{spt}(p)} p(x)u(x) \geq \sum_{x \in \text{spt}(q)} q(x)u(x)$$

Moreover, if $u, v : X \to \mathbb{R}$ satisfy (*), then there exist $a, b \in \mathbb{R}$, with $a > 0$, s.t. $u = av + b$.

Assuming that X is a finite set, i.e. $X = \{x_1, \ldots, x_n\}$, then we can identify $p \in P$ with $p = (p_1, \ldots, p_n) \in \mathbb{R}^n$ and similarly for q. So, condition (*) becomes:

$$\sum_{k=1}^{n} p_k u(x_k) \geq \sum_{k=1}^{n} q_k u(x_k)$$

Given p, probability on X, we can see that $\sum_{k=1}^{n} p_k u(x_k)$ is the *expected value* of the random variable $u : X \to \mathbb{R}$.
So, the meaning of the theorem is the following. To compare a couple of elements $p, q \in P$, one has to see which one will give the highest *expected utility* to the decision maker, given its utility function u. This function u is often referred as the "von Neumann-Morgenstern" utility function of the decision maker. These results appeared for the first time in a fundamental work by Von Neumann and Margenstern in 1944 [NM44].
The last statement of the theorem affirm that a "von Neumann-Morgenstern" utility function is determined only up to a positive affine transformation. One should note the difference with the case of decision making under certainty. In that case, the utility function is determined only up to any strictly increasing transformation.

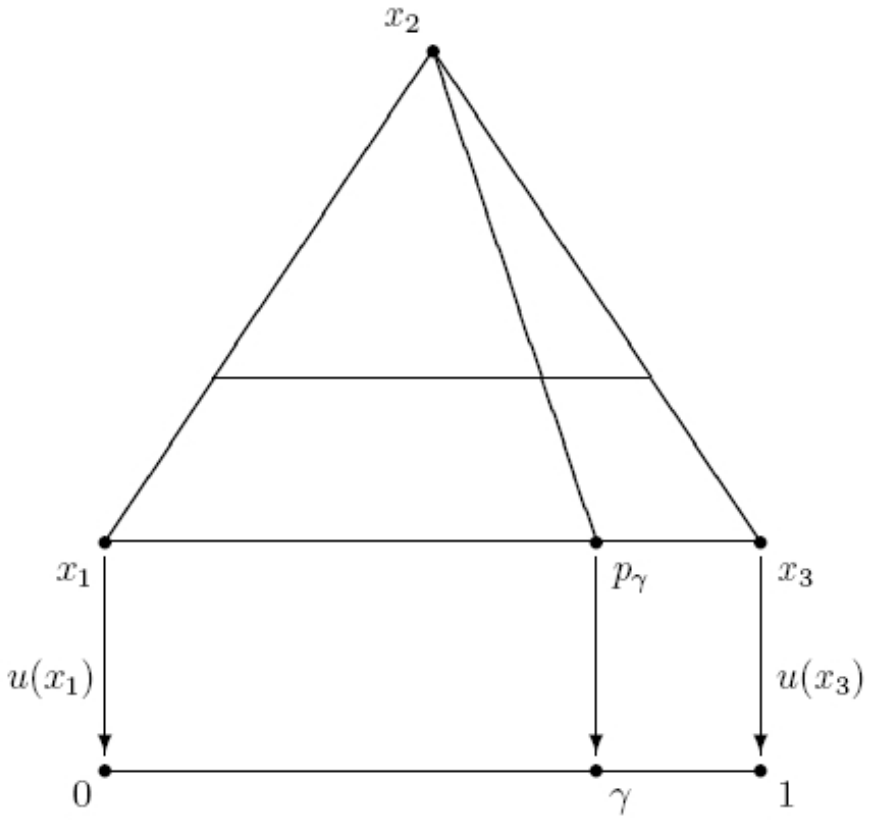

Figure 1.2: The representation theorem.

Hence, in the case of decision making under risk, there is much less freedom in the choice of the utility function.

For a proof of the theorem, one can look at the book of Fishburn[Fis79]. It is possible, however, to give a quick idea of how the proof could be in a very simple case, such as when $X = \{x_1, x_2, x_3\}$. Assume, without loss of generality, that $\delta_{\{x_3\}} \succ \delta_{\{x_2\}} \succ \delta_{\{x_1\}}$. The set P can be identified with a triangle (equilateral) of vertices x_1, x_2, x_3, as seen in Figure 1.2. One can decide to define that $u(x_1) = 0$ and that $u(x_3) = 1$, thus exploiting the freedom in the choice of scale (i.e., fixing the parameters a and b) which is left to us, accordingly to the last statement of the representation theorem.

Then, the value that u will assume on x_2 will be determined by the preferences of the decision maker. The idea is that the archimedean property would assure that

there is a lottery p_γ, concentrated on x_1 and x_3, which is indifferent to x_2. That is:

$$\delta_{\{x_2\}} \sim p_\gamma = \gamma\delta_{\{x_3\}} + (1-\gamma)\delta_{\{x_1\}} = \delta_{\{x_1\}} + \gamma(\delta_{\{x_3\}} - \delta_{\{x_1\}})$$

This is not precisely guaranteed by the archimedean property, as it is stated in the assumption 2: some technical work and the use of the other assumptions are needed to guarantee the existence of this lottery indifferent to x_2. Having p_γ, one simply defines $u(x_2) = \gamma$. So, we have defined $u : X \to \mathbb{R}$. Clearly, there is still a lot of work to be done. In particular, one must guarantee that u really represents (via expected values) the preferences of the decision maker. The key point to assure this is the independence assumption. This assumption says that the behaviour of the preferences of the decision maker on the segment from x_1 to x_3 is replicated in a homothetic way on the chords of the triangle which are parallel to the segment x_1, x_3. Namely, the independence assumption guarantees that, as soon as we know that p_γ is indifferent to x_2, then also all of the lotteries which are located on the segment joining x_2 to p_γ are indifferent to x_2.

A special and useful case to consider is when X is an interval of $\mathbb{R}$. because it allows for applications of the theory to the case in which the final outcomes, i.e. the elements of X, are amounts of money.

Let $u : X \to \mathbb{R}$ be a von Neumann-Morgenstern utility function, which represents the preferences of a decision maker. In the special case that we are considering, i.e. X is an interval of $\mathbb{R}$, there is an important fact that happens. Take $x_1, x_2 \in X$, and consider a lottery involving just x_1 and x_2. We shall have some probability p on x_1, and $1-p$ on x_2. Of course, we can calculate $px_1 + (1-p)x_2$, that we can rewrite as $x_2 + p(x_1 - x_2) =$ (using $\lambda = 1-p$) $= x_1 + \lambda(x_2 - x_1)$. Now, since this is a lottery, we can evaluate the expected utility that the decision maker will get, which is: $u(x_1) + \lambda[u(x_2) - u(x_1)]$. But $x_1 + \lambda(x_2 - x_1)$ is also a real number, which belongs to X, and located somewhere between x_1 and x_2. Since $x_1 + \lambda(x_2 - x_1) \in X$, u is defined on it, so it can be evaluated $u(x_1 + \lambda(x_2 - x_1))$. But the point $x_1 + \lambda(x_2 - x_1)$ can be seen, at the same time, as representing the lottery which assigns probability p to x_1 and $1-p$ to x_2. If we want to see on the graph which is the expected utility that our decision maker associates to this lottery, we have to look at the intercept of the vertical line above $x_1 + \lambda(x_2 - x_1)$ and the segment joining $(x_1, u(x_1))$ and $(x_2, u(x_2))$.

Notice that the real number $x_1 + \lambda(x_2 - x_1)$ represents the *expected gain* coming from the lottery.

What we see in the figure is that our decision maker prefers (to the lottery) to receive *with certainty* an amount of money equal to the expected gain of the lottery.

To see a numerical example, consider a lottery which gives 0 euro with probability $1/2$ and 1000 euro with probability $1/2$. Then, the expected gain is 500 euro.

So, there are two different things that can be considered: the lottery, in which the decision maker will receive either 0 or 1000 euro, with probability $1/2$ each, and another (degenerate) lottery in which the decision maker will receive with certainty (or with probability 1, which we consider equivalent) the amount of 500 euro.
The decision maker could prefer to receive 500 euro with certainty instead of the lottery, thus exhibiting an instance of what is called *risk aversion*. Or could be exhibiting the opposite attitude, which would qualify him as a *risk lover*. There is also the possibility of indifference (risk neutral).
In general, risk aversion means that a decision maker always prefers to receive with certainty the expected value of the lottery to the lottery. Geometrically, risk aversion is equivalent to the *concavity* of u.

Chapter 2

Non-Cooperative Games

Game theory is the mathematical theory of conflict and cooperation situations in which several agents make decisions and an outcome is produced. Every agent has its own preferences over the set of possible outcomes. We have introduced preferences in the previous chapter, so we shall take those concepts for granted.

Games can also be seen as decision problems with several decision-makers; hence, there are many models that can be used to manage these problems, and many different classifications depending on the presence of different aspects. The fundamental classification is given by the possibility of making binding agreements between players. Thus, we can speak of

- Non-cooperative game theory, that deals with maximization of the utility of players

- Cooperative game theory, that deals with allocation of benefits between players

In the next section, we will briefly describe non-cooperative games in two different form: the strategic and the extended ones.

2.1 Strategic-form Games

Let's start introducing the elements that are really involved in a static conflict situation:

- $N = \{1, .., n\}$, the set of players.
- $X = \{X_i\}_{i \in N}$, the strategy sets of players.
- R, the set of possible outcomes.
- A map $f : X \to R$, which assigns to every strategy profile x its corresponding outcome
- $\{\succeq_i\}_{i \in N}$, the preferences of the players (total preorder on R)
- $\{h_i\}_{i \in N}$, the utility functions of the players, representing their preferences on R.

Now, we can define a function $H_i : X \to \mathbb{R}$ such that $H_i(x) = h_i(f(x))$, for all $i \in N$ and all $x \in X$, and say that:

Definition 5 *An n-person strategic game G with player set N is a 2n-tuple*

$$((X_i)_{i \in N}, (H_i)_{i \in N})$$

such that, for all $i \in N$, X_i is the non-empty set of strategies of player i, and H_i is its payoff function, which assign to every strategy $x \in X$ the payoff $H_i(x)$ that player i obtains if such a strategy is played.

This definitions takes into account the elements described before and gives us a simple but realistic model that can be applied to many conflict situation.

A play of a game G always takes place in the same way: there may be some stage in which players can communicate and even make non-binding agreements, then each player i chooses an $x_i \in X_i$. These choices are made *simultaneously* and *independently*. Finally every player i receives his payoff $H_i(x)$.
Given the formal definition of a strategic game, it is obvious that we should be interested in finding the best strategies for each player. This leads to investigate an interesting aspect of game theory: the concept of solution.

2.2 Nash equilibrium in strategic-form games

The most important solution concept for strategic games is the Nash equilibrium, introduced in Nash [Nas51]. Kohlberg [Koh90] says that the main idea under this equilibrium is "to make a bold simplification and, instead of asking how a the process of deduction might unfold, ask where its rest points may be". In fact, a Nash equilibrium of a strategic game is simply a strategy profile such that no player gains when deviating from it; i.e. the Nash equilibrium concept searches for *rest points* of the conflict situation described by the strategic game.

Definition 6 *Let $G = ((X_i)_{i \in N}, (H_i)_{i \in N})$ be a strategic game. A Nash equilibrium of G is a strategy profile $x \in X$ which satisfies that*

$$H_i(x) = H_i(x_1, .., x_i, .., x_n) \geq H_i(x_1, .., x_i^{'}, .., x_n)$$

for all $x_i^{'} \in X_i$ and for all $i \in N$.

Let us make an example to explain the meaning of this definition.

Example 1 *Player I has to choose a strategy in X and Player II has to choose a strategy in Y. Player I has utility function f on the outcomes of the game, while Player II has utility function g. So the game is $G = (X, Y, f, g)$. Let $(\bar{x}, \bar{y}) \in (X, Y)$ be a Nash equilibrium. Let us remember that players choose strategies independently and simultaneously (they do not know each other's move for sure). Moreover, they cannot make binding agreement. Imagine that the two players make an agreement (from which they could always back out) in which they promise to play in this way: Player I chooses strategy $\bar{x}$ and Player II chooses strategy $\bar{y}$. Player I should reason in this way:*
'We agreed to play in that way, but I can violate that agreement and nothing will happen to me, so let's see if I can play a better strategy than $\bar{x}$. There are two possibilities: either Player II doesn't respect the agreement too (so is not useful to take it into account), or he respects it. In the latter case, I could look for a strategy $\underline{\bar{x}}$ such that $f(\underline{\bar{x}}, \bar{y}) > f(\bar{x}, \bar{y})$' Player II should reason in a similar way with role inverted. But neither Player I nor Player II will find such strategies because

$$f(\bar{x}, \bar{y}) \geq f(x, \bar{y}) \text{ for each } x \in X$$

$$g(\bar{x}, \bar{y}) \geq f(\bar{x}, y) \text{ for each } y \in Y$$

as the definition of Nash equilibrium says.

The definition of equilibrium is structured to take into account this considerations: the Nash conditions say that no player has incentives for deviating from the strategy suggested by the equilibrium, *given the fact that the other is not deviating too*! Notice that is easy to find a game in strategic-form without Nash equilibria, as this example shows:

Example 2 *Given the 'matching pennies' game:*

I / II	*P*	*D*
P	*(-1,1)*	*(1,-1)*
D	*(1,-1)*	*(-1,1)*

Let us look for a Nash equilibrium.

- (D, P)*: not an equilibrium because because Player II could get a better payoff playing D.*
- (D, D)*: not an equilibrium because because Player I could get a better payoff playing P.*
- (P, D)*: not an equilibrium because because Player II could get a better payoff playing P.*
- (P, P)*: not an equilibrium because because Player I could get a better payoff playing D.*

Given a strategy profile (x, y) *with* $x, y \in \{P, D\}$*, each player has an incentive to deviate. It is clear that there are no Nash equilibria in this game.*

The last example shows some limitations of Nash equilibrium as a concept of solution for games, in fact we would expect that there exists a solution for at least any finite game. Luckily this problem has been resolved quite easily and naturally extending the definition of strategy in this way:

Definition 7 *Let* $G = ((X_i)_{i \in N}, (H_i)_{i \in N})$ *be a finite game. The mixed extension of* G *is the strategic game*

$$E(G) = ((S_i)_{i \in N}, (K_i)_{i \in N})$$

where, for every player $i \in N$*,*

- $S_i = \left\{ s_i \in \mathbb{R}^{X_i} |\ \forall x_j \in X_i,\ \ s_i(x_j) \geq 0 \ \ and \ \ \sum_{x_j \in X_i} s_i(x_j) = 1 \right\}$,

- $K_i(s) = \sum_{x \in X} H_i(x)s(x), \forall s \in S$, *where* $S = \prod_{i \in N} S_i$ *and* $s(x)$ *denotes the product* $s_1(x_1)...s_n(x_n)$

The mixed extension of finite game only make sense if the player have preferences over the set of lotteries on R (the set of possible outcomes) and satisfy Assumption 1, 2 and 3 of Section 1.2. In such a case, their preferences can be represented by von Neumann and Morgenstern utility functions: this ensures the meaningfulness of using their expected payoffs K_i. It is interesting to note that $E(G)$ is really an extension of G, in the sense that for every player i, every element of X_i (that can be called *pure strategy*) can be obviously identified with an element of S_i (e.g. pure strategy $x_k \in X_i$ can be identified with mixed strategy s where $s(x_j) = 0$ for every $x_j \in X_i$ except for $s(x_k) = 1$). Let's see the interpretation of mixed extension. Basically, it's just the original game with the difference that players, instead of making pure choices between moves, attach probabilities to each move (in other words, they define a lottery on moves) and then 'extract randomly' the move to play. Now we can state the theorem proved by John Nash in his original paper [Nas51]:

Theorem 6 *Let* $E(G) = ((S_i)_{i \in N}, (K_i)_{i \in N})$ *be a mixed extension of a finite game, then* $E(G)$ *has at least one Nash equilibrium.*

2.3 Incomplete Information Games

Until now we have always treated games of *complete information*, where we assumed that the characteristics of the game are *common knowledge* among all the players. It means that the players know how many they are and what are each other's preferences and utility functions. This assumption is sometimes not very realistic. For example, in an auction of paintings, each bidder has an evaluation of the masterpiece that he is trying to buy, but this is a private information that is not known to all bidders. Moreover, he does not know the value that the other bidders are attributing to that painting. Obviously, it is necessary to introduce another model to manage these kind of games, and this has been done by Harsanyi[Har68]. This kind of situation is called *incomplete information* because there are some aspects of the game that are not known to the players. In our discussion, we will limit to consider situations where only the utility functions of the players are not common knowledge. Harsanyi (see also Myerson [Mye91]) shows that many other kinds of imperfect information situation can be reduced to this one. The basic idea is the use of 'type' of players, where we store all the information about a player that is not known to the others.

Definition 8 *Given the set of players $N = \{1, .., n\}$, a Bayesian game is a 4n-tuple*

$$((A_i)_{i\in N}, (T_i)_{i\in N}, (p_i)_{i\in N}, (u_i)_{i\in N})$$

- T_i *with* $i \in N$*, the set of possibile types of player* i
- A_i *with* $i \in N$*, the set of actions that player* i *could play*
- $p_i : T_i \to \Delta(T_{-i})$ *with* $i \in N$*, a probability-valued function for player* i
- $u_i : A \times T \to \mathbb{R}$*, the utility function of player* i

where $T_{-i} = (T_1, .., T_{i-1}, T_{i+1}, .., T_n)$ is the set of all possible combination of types of all players except i, $T = (T_1, .., T_n)$ is the set of all possible combination of types of all players and $A = (A_1, .., A_n)$ is the set of all possible combination of action of all players

The introduction of a set of types for each player is needed to model the fact that player i does not know who are his opponents, but he knows who *could be*: he has a set of possible 'personalities' for any opponent among which there is the 'actual'one.
In this model, each player has a type that is known only to himself, and he knows what are the probabilities that the other players have a certain combination of types. That is, for any possible type $t_i \in T_i$, the probability-valued function specifies a distribution $p_i(\cdot|t_i)$ over T_{-i} (the set of all possible combinations of types for all players except i), representing what player i would believe about the other players' types if his own type were t_i. Thus, given type t_i, $p_i(t_{-i}|t_i)$ denotes the subjective probability that player i assigns to the event that the other players have types $t_{-i} = (t_1, ..t_{i-1}, t_{i+1}, .., t_n)$.
The interpretation of u_i is that the game specifies a payoff for player i that depends on the combination of action of all the players but also on their types, in fact we expect that different types of players have different preferences over the set of outcomes of the game.
When we study a Bayesian game, we assume that each player i knows the entire structure of the game (as defined above) and his own actual type $t_i \in T_i$ and this fact is common knowledge among all the players in N. Because of this reason, we referred to the objects of choice in a Bayesian game as actions rather than a strategies. In fact, an action is a move that a player i of specific type t_i can choose, while a strategy is a complete plan of actions, one for *each* $t_i \in T$. Thus, a strategy is a function $s_i : T_i \to A_i$.

Belief consistency

The probability-valued function p_i introduced in the last section represents what player i would believe about the other players' types: in other words, it represents his *beliefs*. We say that beliefs in a bayesian game are *consistent* iff there is some common prior distribution over the set $T = (T_1, .., T_n)$ of type combination such that each players' beliefs are just the conditional probability distribution that can be computed from the prior distribution by Bayes' formula (used only whenever the marginal distribution is positive). More formally,

Definition 9 *Given a bayesian game Γ, the beliefs p_i are consistent iff there exists some probability distribution $P \in \Delta(T)$ such that*

$$p_i(t_{-i}|t_i) = \frac{P(t_1,..t_n)}{\sum_{(s_{-i})\in T_{-i}} P(s_1,..s_{i-1},t_i,s_{i+1},..,s_n)} \qquad \forall (t_1, ..t_n) \in T, \quad \forall i \in N$$

where $s_{-i} = s_1, ..s_{i-1}, s_{i+1}, .., s_n$

Most of the bayesian games that have been studied in applied game theory have consistent beliefs. One reason for this tendency is that consistency simplifies the definition of the model: it is simpler to specify one probability distribution than many functions that depend on types. Furthermore, inconsistency often seems an unnatural feature for a model. In a consistent model, different beliefs among player can be explained by differences in information, whereas inconsistent beliefs involve differences in opinions among players that cannot be derived from anything but must be assumed a priori.

Type-agent representation

Harsanyi [Har68] discussed a way to represent any Bayesian game by a game in strategic form, which can be called *type-agent representation*. In this representation there is a *player-agent* for every possible type of every player in the Bayesian game. It is necessary to assume that the set T_i are disjoint, so that $T_k \cap T_j = \oslash$.

Definition 10 *Given a Bayesian game $\Gamma_b = ((A_i)_{i\in N}, (T_i)_{i\in N}, (p_i)_{i\in N}, (u_i)_{i\in N})$, a type-agent representation of Γ_b is a strategic form game defined as*

$$G(\Gamma_b) = ((D_r)_{r\in T^*}, (v_r)_{r\in T^*})$$

where

- $T^* = \bigcup_{i\in N} T_i$ *is the set of player-agents*
- $D_{t_i} = A_i \quad \forall i \in N \forall t_i \in T_i$ *is the set of strategies of player-agent t_i*
- $v_{t_i} : \times_{s\in T_*} D_s \to \mathbb{R}$ *is the utility function of player-agent t_i defined as*

$$v_{t_i}(d) = \sum_{t_{-i} \in T_{-i}} p(t_{-i}|t_i) u_i((d(t_j))_{j \in N}, (t_j)_{j \in N})$$

where $t_{-i} = (t_1, .., t_{i-1}, t_{i+1}, .., t_n)$ *and*
$(d(t_j))_{j \in N} = (d(t_1), .., d(t_{i-1}), d(t_i), d(t_{i+1}), .., d(t_n))$

The idea is that player-agent t_i is responsible for selecting the action that player i will use if he has type t_i. Thus, the strategy profile d is a plan that define an action for every type of every player i in the Bayesian game Γ_b, while the payoff for player-agent t_i is the conditionally expected utility payoff for player i, given that he has type t_i.

2.4 Bayesian equilibrium

For a game of incomplete information, Harsanyi [Har68] defined a *Bayesian equilibrium* to be any Nash equilibrium of the type-agent representation in a strategic form (as defined in Section 2.3).That is, a Bayesian equilibrium ispecifies an action for each type of each player, such that each player would be maximizing his own expecting utility when he knows his own type but does not kknow the types of the other players.
It is necessary to specify what every possible type of player would do, not just the actual type, because otherwise we could not define the expected utility payoff for each player, who does not know the other players' actual types.
Before we formally state the definition of Bayesian equilibrium, we must introduce another definition:

Definition 11 *Let* $\Gamma^b = ((A_i)_{i \in N}, (T_i)_{i \in N}, (p_i)_{i \in N}, (u_i)_{i \in N})$ *be a Bayesian game, a mixed strategy profile is any* $\sigma \in \times_{i \in N} \times_{t_i \in T_i} \Delta(A_i)$ *such that*

- $\sigma = ((\sigma_i(a_i|t_i))_{a_i \in A_i})_{t_i \in T_i, i \in N}$,
- $\sigma_i(a_i|t_i) \geq 0, \forall a_i \in A_i, \forall t_i \in T_i, \forall i \in N$ *and*
- $\sum_{a_i \in A_i} \sigma_i(a_i|t_i) = 1, \forall t_i \in T_i, \forall i \in N$

In such a mixed strategy profile σ, the number $\sigma_i(a_i|t_i)$ represents the conditional probability that player i would use action a_i if his type were t_i. The mixed strategy for type t_i of player i is $\sigma_i(\cdot|t_i) = (\sigma(a_i|t_i))_{a_i \in A_i}$.
Definition 11 makes possible the introduction of the main concept of solution for a Bayesian game:

Definition 12 *A Bayesian equilibrium of the game Γ_b is any mixed-strategy profile σ such that, $\forall i \in N$ and $\forall t_i \in T_i$*

$$\sigma_i(\cdot|t_i) \quad \in \quad \arg\max_{\tau \in \Delta(A_i)} \sum_{t_{-i} \in T_{-i}} p_i(t_{-i}|t_i) \sum_{a \in A} [\prod_{j \in N-i} \sigma_j(a_j|t_j)] \tau_i(c_i) \, u_i(c,t)$$

Chapter 3

Introduction to Auction Theory

People with objects to sell usually want to sell them for the highest price they can get. Sometimes there is no choice but bargain the price with the possible purchasers, sometimes there is an established market that decides the price with rules of his own; in situations where the seller is a monopolist, he can decide the set of rules by which the price will be fixed. Such set of rules is called an *auction mechanism*. Moreover, it is necessary to specify how the prospective buyers evaluate the goods, and this could be done defining a a *value model*. Auction theory studies these mechanism and models, analyzes behaviors and outcomes of sellers and buyers in order to determine the properties of a mechanism and its efficiency in certain situations. Auctions are certainly one of the most old and known forms of sales of goods, but only recently the analysis of games of incomplete information has permitted a compared analysis of the efficiency of different auction mechanisms and a deeper knowledge on designing optimal mechanisms. In this chapter we will introduce the basic value models and auction mechanisms together with some important results of auction theory.

3.1 Standard auction mechanisms

The typical situation where an auction is suitable to allocate some goods can be described in this way: on a side of the market (usually, the offering side) a monopolist wants to sell some goods; on the other side there are two or more potential buyers. It is implicitly assumed that the monopolist will choose the procedure (or mechanism) to allocate the goods, but this does not necessarily mean that he can extract the entire surplus, because he does not know the buyers' true valuation of the goods. It is useful to give a classification of the most common auction mechanisms to understand better their characteristics. The first distinction can be made between *open* and *sealed-bid* auctions.
In the open auction mechanisms, the seller announces prices or the bidders call out the prices themselves, thus it is possible for each agent to observe the opponents' moves. The most common type of auction in this class is the *ascending* (or *English*) auction, the well-known procedure typical of artwork auctions, where the price is successively raised until no one bids anymore and the last bidder wins the object at the last price offered. Another diffused type, the *descending* or *Dutch* auction works in exactly the opposite way: the auctioneer starts at a high price and then lowers it continuously. The first bidder that accepts the current price wins the object at that price. This type of auction gets his name from the Dutch flower auctions, where these mechanism is implemented replacing the auctioneer with an electronic clock that decreases the original price of a given percentage every fixed step of time.
The sealed-bid auction mechanism are characterized by the fact that offers are only known to the respective bidders (as the name suggest, offers are submitted in sealed envelopes). In the *first-price* sealed-bid auction each bidder independently submits a single bid without knowing the others' bid, and the objects is sold to the bidder who made the best offer. First-price auction are especially used in government contract. Another widely used and analyzed auction in this class is the *second-price* sealed-bid auction, that works exactly as the first-price one except that the winner pays the second highest bid. This auction is sometimes called Vickrey auction after William Vickrey, who wrote the seminal paper on auctions[Vic61].

3.2 Basic models of auction

Any theoretical conclusion on the effectiveness of different mechanisms depends strongly on the assumptions that are made on the bidders' evaluations of the offered goods. These assumptions are included in models that try to mimic some real situations. A key feature of auctions is the presence of asymmetric information,

thus we expect that the mathematical formalization can be done with incomplete information games.

In the basic *independent private-value* model each bidder:

- knows privately the actual value of the object to himself
- believes that the other bidders' evaluation of the object can be described by a probability distribution that is identical for all the bidders
- believes that there is statistical independence between the individual evaluation.

This model implies that any difference in the bidders' evaluation depends on individual characteristics of each bidder. An auction that uses this model can be formalized with a Bayesian game where each player's type is his value for the object.

Example 3 *Consider a first-price auction for a single indivisible object with independent private values. There are n bidders and each bidder has a evaluation $v_i \in [0, M]$. Each bidder consider the values to the others $n-1$ bidders to be independently drawn from the interval $[0, M]$ with a specified cumulative distribution F. The rules of the auction are:*

- *each bidder i simultaneously submits a sealed bid b_i*
- *the object is delivered to the bidder whose bid is highest*
- *the winner pays the amount of his bid.*

It is clear that is possible to formalize the situation described above with a Bayesian game The bidders are players in the set $N = \{1, .., n\}$. The set of possible types of each player is the interval $[0, M]$. Each player i has a type $v_i \in [0, M]$ that is his private value for the object. The probability-valued function p_i can be easily extracted from the distribution F. Thus, letting $b = (b_1, ..., b_n)$ denote the profile of bids and $v = (v_1, ..., v_n)$ denote the profile of types to the n players, the expected payoff to player i is

$$\begin{aligned} u_i(b, v) &= v_i - b_i \quad \text{if } \{i\} = \arg\max_{\{1,...,n\}} b_j, \\ &= 0 \qquad\quad \text{if } \{i\} \notin \arg\max_{\{1,...,n\}} b_j, \end{aligned}$$

In the opposite model, the *pure common-value* model, the actual value is the same for everyone, although the bidders may have different estimates of the value because they may have private information about the its quality. This is the case, for example, of the sales of public resources to companies with the same level of

technology (hence, that have the same prospects of profits) but have different evaluations of what will be the working conditions that they are going to face. This model has been introduced by Wilson in a paper dated 1969 [Wil69].

Example 4 *Consider a first-price auction for a single indivisible resource with an unknown value in a pure common-value model where only two bidders participate. The resource's monetary value depends on three independent random variables* x_0,x_1,x_2*, each drawn from a uniform distribution on the interval* $[0, 1]$*. The winner will get an amount of money equivalent to* $A_0x_0 + A_1x_1 + A_2x_2$ *derived from the exploitation of the resource, where* A_0,A_1,A_2 *are given non-negative constants that are known to the bidders. Before the auction takes place, the two bidders have analyzed the resource to extract useful informations. Bidder* 1 *observed signals* x_0 *and* x_1*, while bidder* 2 *observed signals* x_0 *and* x_2*. Bidder* 1 *does not know* x_2 *and bidder* 2 *does not know* x_1*. The rules of the auction are:*

- *each bidder* i *simultaneously submits a sealed bid* c_i
- *the object is delivered to the bidder whose bid is highest*
- *the winner pays the amount of his bid*
- *in case of a tie, each bidder has a* $1/2$ *probability of getting the object at the bidded price.*

Assuming that both bidders are risk-neutral, we can view this conflict situation as a Bayesian game where the bidders are players, their types are (x_0, x_1) *for player* 1 *and* (x_0, x_2) *for player* 2*, the set of action is the interval of possible bids (presumably* $[0, T]$ *where* $T = A_0x_0 + A_1x_1 + A_2x_2$*), and their utility function is*

$$\begin{aligned} u_i(c_1, c_2, (x_0, x_1), (x_0, x_2)) &= A_0x_0 + A_1x_1 + A_2x_2 - c_i && \text{if } c_i > c_j \\ &= (A_0x_0 + A_1x_1 + A_2x_2 - c_i)/2 && \text{if } c_i = c_j \\ &= 0 && \text{if } c_i < c_j \end{aligned}$$

A more general model encompassing both the private-value and the common-value model as special cases assumes that each bidder has some private information on the object, but each bidder's value depends on *all* private signals of the bidders. That is, bidder i has a private signal t_i and his value of the object is $v_i(t_1, ..., t_n)$ if all signal were available to i. For example, the evaluation of a painting is composed by the bidder's private evaluation (how much pleasure the bidder would get from owning it) and by the other's private evaluation (how much the other bidders would like to get the painting), because it affects what the resale value can be. This model has been introduced in a important paper by Milgrom and Weber [MW82].

3.3 Bayesian equilibrium in auction mechanisms

A deeper analysis reveals that, beyond different realization of the auctions, there is a strategical equivalence between the Dutch auction and the first-price auction. In fact both procedures, represented by normal form games, have the same set of strategies and the same payoff utility function: both cases expect that the bidders call their price without knowing anything on the others' behavior, and that the buying price will be the highest bid. Thus, the first-price auction and the Dutch one are only two different ways of implementing the same mechanism of goods allocation. Hence we can focus on the other three mechanisms and not consider the Dutch one. It is useful to analyze these games and look for Bayesian equilibria in order to compare different auction mechanisms. Let us find the equilibria for the private-value first-price auction examples of the previous section:

Example 5 *(Referring to Example 3) Suppose that there is a Bayesian equilibrium in which every player decides his bid calculating $\beta(v_1)$ where β is a differentiable and increasing function. Suppose also that the types are uniformly distributed, with $F(x) = x/M$ for any $x \in [0, M]$.*
In the equilibrium, each player i knows that the other players will not bid more than $\beta(M)$, so he will never submit a bid higher than $\beta(M)$. Let us say that player i submit a bid $b_i = \beta(w_i)$. The probability that another player j will submit a bid that is less than b_i is v_i/M. In fact, j will submit a $b_j < b_i$ if and only if $\beta(v_j) < \beta(w_i)$, and this can happen only if $v_j < w_i$, because β is increasing. Hence, bidder i has a probability of $(v_i/M)^{n-1}$ of winning the object because the other $n-1$ bidders' types are independently and uniformly distributed. Thus, the expect payoff to player i bidding $\beta(w_i)$ with value v_i is

$$u(\beta(w_i), v_i) = (v_i - \beta(w_i))(\tfrac{v_i}{M})^{n-1}$$

However, as we assumed before, $\beta(v_i)$ is the optimal bid for player i: this implies that the derivative $u_i'(\beta(v_i), v_i)$ should be equal to zero, that is

$$(v_i - \beta(v_i))\tfrac{1}{M}(n-1)(\tfrac{v_i}{M})^{n-2} + (1 - \beta'(v_i))(\tfrac{v_i}{M})^{n-1} = 0$$

This differential equation implies that

$$\beta(x)(\tfrac{x}{M})^{n-1} = \textstyle\int_0^x y(n-1)(\tfrac{y}{M})^{n-2}(\tfrac{1}{M})dy$$

Resolving the integral, the previous equation becomes

$$\beta(x) = (1 - \tfrac{1}{n})x$$

Because the bidder who wins the auction pays for the object the equivalent of his bid, the expected revenue of the seller will be $(1 - \frac{1}{n})\overline{v}$*, where* $\overline{v}$ *is the winning bid. Notice that the margin of profit of each player in this Bayesian equilibrium depends on the number of players admitted, this implies that the higher is the number of players, the higher will be the revenue of the seller.*

It can be demonstrated that the Bayesian equilibrium of a first-price auction with private value can be obtained if the players choose their bids according to a function

$$\beta(x) = \frac{\int_0^x y(n-1)F(y)^{n-2}F'(y)dy}{F(x)^{n-1}}$$

Griesmer, Levitan and Shubik[GLS67] analysed the equilibria of first-price auction mechanism when contestants' valuations are drawn from uniform distributions with different supports.
Let us look for equilibrium in a second-price auction in a private-value model:

Example 6 *Consider a second-price auction with private values for a single indivisible object. There are* n *potential buyer, each buyer* $i \in [1, n]$*'s evaluation of the object is* v_i *and that the index* i *indicates the order of arrival of the sealed envelopes. The rules of the auction are as follows:*

- *each bidder submits his offer in a sealed envelope with no information on the other offers*
- *the object is assigned to the bidder with the lowest index* i *among those that have submitted the highest bid*
- *the winner pays the highest bid made by one of the other players.*

This game is similar to the one of Example 5, except for the utility function that now is:

$$\begin{array}{ll} u_i(b, v) = v_i - max\{b_j | j \in N \backslash i\} & \text{if } \{i\} = \arg\max_{\{1,\dots,n\}} b_k, \\ = 0 & \text{if } \{i\} \notin \arg\max_{\{1,\dots,n\}} b_k, \end{array}$$

It is easy to verify that there is a truth-telling equilibrium, that is that each player i *bids his own evaluation* v_i*. To confirm this, suppose that the other players' best offer his* w*. Suppose also that* i *offers* $v_i - x$*. If* $v_i - x > w$ *then* i *gets the object at a price* w *with a payoff of* $u_i = v_i - w$*, exactly as if he bidded* v_i*; if* $v_i - x < w$ *and* $v_i > w$*, player* i *do not get the object even if he has the highest evaluation, obtaining a null payoff instead of a better* $v_i - w$ *if he played* v_i*. Suppose instead that player* i *makes a bid of* $v_i + x$*. If* $v_i + x > w > v_i$ *then* i *gets the object at*

a price w with a payoff of $u_i = v_i - w < 0$, while if he offered v_i he would have obtained a payoff of 0; if $v_i + x < w$, player i do not get the object exactly as if he offered v_i. We can conclude that each player do not get a better payoff playing a strategy different from declaring his own evaluation, hence $(v_1, ..v_n)$ is a Nash equilibrium. Moreover, the revenue of the seller is equivalent to the second-best offer in the auction.

In an English auction under private-value hypothesis, the best strategy for any player i is to remain in the competition, making small raisings, until the price reaches his evaluation v_i, and then drop out: this means that the good is assigned to the bidder with the highest valuation, and the price to pay will be equivalent to the second highest valuation (plus a small delta). Comparing the second-price auction and the English one with independent private values, we can realize that, from the point of view of the seller, are equivalent: both assign the object to the bidder who values it the most and both set the price to the second-best evaluation among the bidders. However, this equivalence applies only for private values, or if there are just two bidders: with any common components to valuations and more than two bidders, when some player quits, he implicitly reveals some informations on the value of the object that condition the other players' behavior.

3.4 Revenue Equivalence Theorem

Let us start with a definition:

Definition 13 *An auction mechanism is efficient if and only if the offered object is always given to the buyer with the highest valuation for it.*

It is easy to verify that all the four basic mechanisms shown in Section 3.1 are efficient. Assuming that an auction mechanism is efficient, an interesting problem is to establish which procedure can guarantee the maximum revenue to the seller. As we hinted in the previous section, economic theory provided some fundamental and surprising results on the equivalence (at equilibrium) of the expected revenues of various auction mechanisms. Vickrey provided the earliest conceptualization and results in his famous paper [Vic61], which was, together with another paper [Vic62], a major factor in his 1996 Nobel prize. In the same year, Myerson [Mye81] and Riley and Samuelson [RS81] showed that Vickrey's results apply very generally.

Theorem 7 . ***(Revenue equivalence theorem)****. Assume that*

- *there are N risk-neutral potential buyers*
- *the independent private-value model is used*
- *the buyers are symmetric, that is, they cannot be distinguished one from the other*

then all the efficient auction mechanisms guarantee to the seller the same expected revenue, and each bidder makes the same expected payment as a function of his valutation.

This theorem implicitly defines a wide class of equivalence of auction mechanisms (in terms of expected revenue) and both the first-price and the second-price sealed auction belong to this class. This could be surprising: in the first-price sealed auction, the winner pays the the price that he called while in the second-price one the winner pays a price equal to the highest bid made by the other players. The fact is that the players' best strategy in the second-price auction is to bid their true valutation while in the first-price auction the bidders face a trade-off between lowering the offer (thus obtaining a better payoff in case of success) and getting higher probability of success (but paying more for the object). As we saw in the previous section, it is optimal for a bidder in a first-price auction to bid his valutation minus a discount: the revenue equivalence theorem states that this discount compensate exactly (in expected value) the reduction of payment caused by the second-price mechanism.

Chapter 4

DCaseLP and Jade

4.1 Introduction

DCaseLP [Mig02, AMMM02] means ***D****istributed CaseLP*, where **CaseLP** [MMZ99, BDM+99] is the acronym for ***C****omplex* ***A****pplication* ***S****pecification* ***E****nvironment based on* ***L****ogic* ***P****rogramming*.
DCaseLP is a software environment designed and developed by the Logic Programming Group at the Department of Computer Science of the University of Genova in Italy, with the aim of providing a development tool for specifying, implementing, executing and debugging prototypes of Multi-Agent Systems (abbreviated with MASs).
DCaseLP has originated from the intention of overcoming the deficiencies present in a previously developed software tool, namely CaseLP, which provides a prototyping method and a set of tools and languages to support the realization of prototypes of complex applications.
Correctness and reliability of a developed software are difficult to be guaranteed, particularly for distributed software systems where a set of entities have to cooperate and coordinate in order to exchange information. Moreover, many distributed systems must use existing software modules and, consequently, must integrate information from a potentially large number of different sources.
Thus, integration and reuse of different kinds of information and software tools constitute an urgent need that new software products must satisfy. CaseLP came into being with the purpose of helping the developers of MASs to accomplish these tasks.

JADE (Java Agent DEvelopment Framework) [Til] is a middleware for the development of distributed multi-agent applications: it is a free software distributed in

open source under the terms of the LGPL (Lesser General Public License Version 2) by TILAB, the copyright holder.
JADE is a software development framework aimed at developing multi-agent systems and applications conforming to **FIPA** (Foundation for Intelligent Physical Agents) standards for intelligent agents. It includes two main products: a FIPA-compliant agent platform and a package to develop Java agents. It also offers a set of graphical tools to support the debugging and deployment phases.
JADE has been fully coded in Java and an agent programmer, in order to exploit the framework, should code his/her agents in Java, following the implementation guidelines described in the programmer's guide [BCTR04].
In the first part of this chapter, we will describe the JADE platform and some of the FIPA specifications, while in the second part we will outline the advantages and disadvantages of CaseLP first, and then of the first release of DCaseLP, proceeding toward the current release of DCaseLP, while in the second one we will describe the features of the JADE platform .

4.2 The JADE platform

JADE (Java Agent DEvelopment Framework) is a software framework fully implemented in Java language. It simplifies the implementation of multi-agent systems through a middleware compliant with FIPA specifications and through a set of tools that supports the debugging and deployment phase.
The agent platform can be distributed across machines (which not even need to share the same OS) and the configuration can be controlled via a remote GUI. The configuration can be even changed at run-time by moving agents from one machine to another one, as and when required. The only system requirement is the Java Run Time version 1.2.
JADE is written in Java language and is made of various Java packages, giving application programmers both ready-made pieces of functionality and abstract interfaces for custom, application dependent tasks. Java was the programming language of choice because of its many attractive features, particularly focused on object-oriented programming in distributed heterogeneous environments; some of these features are Object Serialization, Reflection API and Remote Method Invocation (RMI).
The communication architecture offers flexible and efficient messaging, where JADE creates and manages a queue of incoming ACL messages, private to each agent; agents can access their queue via a combination of several modes: blocking, polling, timeout and pattern matching based.

The full FIPA communication model has been implemented and its components have been clearly divided and fully integrated: interaction protocols, envelope, ACL, content languages, encoding schemes, ontologies and, finally, transport protocols. The transport mechanism is easily adaptable to any situation, by transparently choosing the best available protocol: Java RMI, event-notification, and IIOP are currently used, but more protocols can be easily added and integration of SMTP, HTTP and WAP has been already scheduled. SL and agent management ontology have been implemented already, as well as the support for user-defined content languages and ontologies that can be implemented, registered with agents, and automatically used by the framework.
JADE is being used by a number of companies and academic groups, both members and non-members of FIPA, such as BT, CNET, NHK, Imperial College, IRST, KPN, University of Helsinky, INRIA, ATOS and many others. It has been recently made available under Open Source License.

4.2.1 JADE architecture

The standard model of an agent platform, as defined by FIPA, is represented in Figure 4.1.

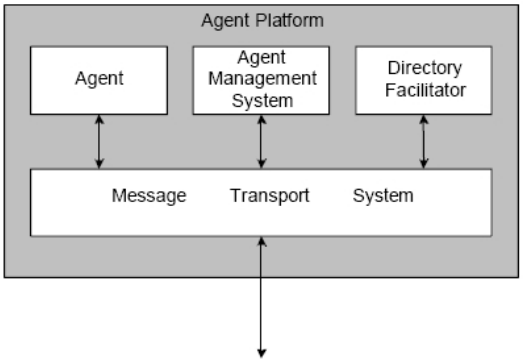

Figure 4.1: Architecture of JADE agent platform

The Agent Management System (AMS) is the agent who acts as supervisor and manages access and use of the Agent Platform. Only one AMS will exist in a single platform. The AMS provides white-page and life-cycle service, maintaining a directory of agent identifiers (AID) and agent state. Each agent must register with an AMS in order to get a valid AID. The Directory Facilitator (DF) is the agent who provides the default yellow page service in the platform. The Message Transport System is the software component managing all the exchange of messages within the platform, including communications with remote platforms: RMI is used for intra-platform communication, whereas CORBA or HTTP are used for inter-platform communication.

When a JADE platform is launched, the AMS and DF are immediately created and the Message Transport System module is set to allow message communication.

The agent platform can be split on several hosts. Only one Java Virtual Machine (JVM) is executed on each host and acts as a basic container of agents, providing a complete run time environment for agent execution and allowing agent concurrency on the same host.

The main-container is the agent container where the AMS and DF lives and where the RMI registry, that is used internally by JADE, is created. The other agent containers, instead, connect to the main container and provide a complete run-time environment for the execution of any set of JADE agents.

4.2.2 Communication in the JADE platform

JADE is a FIPA-compliant platform, so in this section we will describe generally the standards proposed by FIPA regarding Agent Communication.
A *message* is a communicative act of an agent toward one or more other agents. In multi-agent systems often message exchanges are complex and the simple request-response protocol, typical of the client-server model, is often not sufficient. Agents communicate by exchanging messages that represent speech acts, and which are encoded in an Agent Communication Language (ACL).
Agent communication is compounded of three fundamental aspects: the structure of the messages, their representation and their transport. FIPA introduced various Agent Communication specifications that refer to different aspects of communication, and are the following:

- Communicative Acts
- Content Languages
- Interaction Protocols

Communicative Acts The Communicative Acts (CAs) specifications define the communication language and its semantics. These formal definitions provide a clear and unambiguous way for expressing the standardized meaning of the communicative actions carried out by the agents, mainly the messages they exchange and the protocols they use. For more details, see the FIPA Communicative Act Library Specification downloadable from http://www.fipa.org/specs/fipa00037/ .
Content Languages The Content Language (CL) specifications deal with different representations of the content of messages and describe the Semantic Language. The Semantic Language is used to define the semantics of FIPAs Agent Communication Language (ACL). For more details, see the FIPA SL Content Language Specification downloadable from http://www.fipa.org/specs/fipa00008/.
Interaction Protocols The Interaction Protocols (IPs) specifications are message exchange protocols for ACL messages. They constitute a frameworks for the communicative acts defined in the CAs specifications. Every IP specification describes a conversation between agents: the flow of messages exchanged by the participants. Various IP specifications are viewable at http://www.fipa.org/repository/ips.php3.
The *ACL message* is the entity that is transferred among agents: the description of its structure can be found in FIPA ACL Message Structure Specification (downloadable from http://www.fipa.org/specs/fipa00061/). To realize a communication, agents must share a common vocabulary in order to understand each other and interpret the meaning of the terms, objects, relations in the same manner. An *ontology*

actually provides such a vocabulary. Through an ontology, a group of agents can communicate an represent knowledge regarding some topic and can define a set of relationships and properties on the entities contained in the ontology.
An ACL message contains the message, called content, together with additional information like the type of communicative act that it represents (called *performative*), the sender and receivers and the ontologies for the interpretation of the content.
The content of an ACL message must be expressed using a content language, but the receivers needs also at least one ontology to understand the meaning of the content. Clearly, the application domain determines the effective parameters used in the agent communication, the only necessary parameter in all the messages is the performative.The messages transmitted within specific implementations can include user-defined parameters in addition to the ones proposed by FIPAs standard.

4.2.3 Using the JADE platform

To support the difficult task of debugging multi-agent applications, some tools have been developed. Each tool is packaged as an agent itself, obeying the same rules, the same communication capabilities, and the same life cycle of a generic application agent.
Remote Monitoring Agent
The *Remote Monitoring Agent* (RMA) allows controlling the life cycle of the agent platform and of all the registered agents. The distributed architecture of JADE allows also remote controlling, where the GUI is used to control the execution of agents and their life cycle from a remote host.
An RMA is a Java object, instance of the class `jade.tools.rma.rma` and can be launched from the command line as an ordinary agent (i.e. with the command java `jade.Boot myConsole:jade.tools.rma.rma`), or by supplying the `-gui` option to the command line parameters (i.e. with the command java jade.Boot .gui). More than one RMA can be started on the same platform as long as every instance has a different local name, but only one RMA can be executed on the same agent container.

The followings are some of the commands that can be executed from the toolbar of the RMA GUI, for a detailed explanation refer to the JADE Documentation. The listed commands are performed by using the current selection of the agent tree as the target.

- **Start New Agent.** This action creates a new agent. The user is prompted for the name of the new agent and the name of the Java class the new agent is an

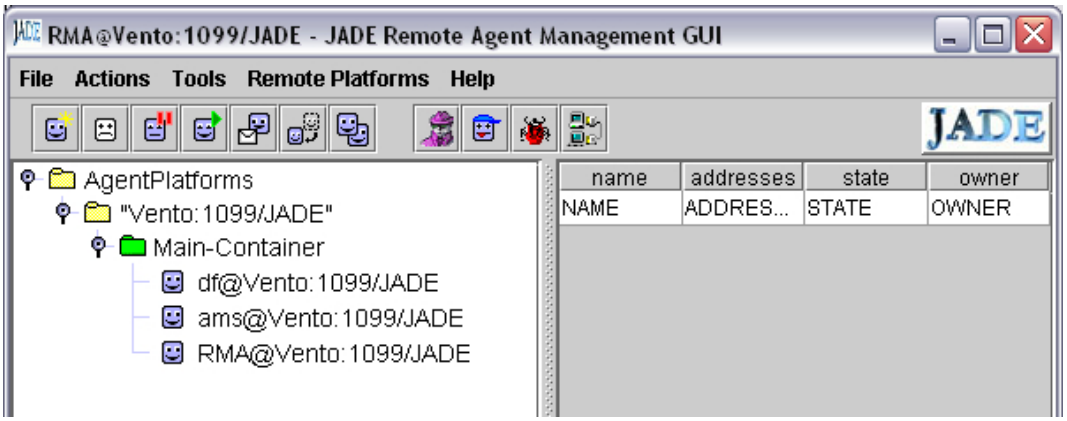

Figure 4.2: GUI of the RMA agent.

instance of. If the class of the agent is `tuPInJADE.JShell42PCycleGui`, then a 'File Open' window will appear and a the tuProlog theory file containing the agent's code must be selected. Moreover, if an agent container is currently selected, the agent is created and started on that container; otherwise, the user can write the name of the container he wants the agent to start on. If no container is specified, the agent is launched on the Agent Platform Main- Container.

- **Kill Selected Agents.** This action kills all the agents and agent containers currently selected. Killing an agent container kills all the agents living on the container and then de-registers that container from the platform. Of course, if the Agent Platform Main-Container is currently selected, then the whole platform is shut down.

- **Suspend Selected Agents.** This action suspends the selected agents. Beware that suspending a system agent, particularly the AMS, deadlocks the entire platform.

- **Resume Selected Agents.** This action puts the selected agents back into the active state, provided they were suspended.

- **Send Custom Message to Selected Agents.** This action allows to send an ACL message to an agent. When the user selects this menu item, a special dialog is displayed in which an ACL message can be composed and sent.

- **Migrate Agent.** This action allows to migrate an agent. When the user selects this menu item, a special dialog is displayed in which the user must specify

the container of the platform where the selected agent must migrate. Not all the agents can migrate because of lack of serialization support in their implementation. In this case the user can press the cancel button of this dialog.

- **Clone Agent.** This action allows to clone a selected agent. When the user selects this menu item a dialog is displayed in which the user must write the new name of the agent and the container where the new agent will start.

Agent Identification. According to the FIPA specifications, each agent is identified by an Agent Identifier (AID). An Agent Identifier (AID) labels an agent so that it may be distinguished unambiguously within the Agent Universe.
The AID is a structure composed of a number of slots, the most important of which is *name*. The name parameter of an AID is a globally unique identifier that can be used as a unique referring expression of the agent. JADE uses a very simple mechanism to construct this globally unique name by concatenating a user-defined nickname to its home agent platform name (HAP), separated by the '@' character. Therefore, a full valid name in the agent universe, a so-called GUID (Globally Unique Identifier), is `foglia@Vento:1099/JADE` where `foglia` is the agent nickname that was specified at the agent creation time, while `Vento:1099/JADE` is the platform name. Only full valid names should be used within ACLMessages.
Debugging features JADE offers two different agents that provides interfaces to debug the MAS: the Sniffer Agent and the Inspector Agent.
The *Sniffer Agent* is an agent 'enriched' with so-called sniffing features: it displays the messages exchanged by agents selected by the user, as a sort of sequence diagram. The Sniffer Agent can be started from the RMA GUI clicking on the Tools menu and then selecting the Start Sniffer item from the displayed menu (shown in Figure 4.3).
This agent offers an ad-hoc GUI from which the user can sniff an agent or a group of agents belonging to the MAS: every message sent/received to/from the target agent or group is observed and displayed in the GUI (see Figure 4.4).
In the window on the right of the GUI in Figure 4.4, the user can view the sniffed agents and the flow of messages between them: the box labeled Other represents all the agents of the platform that are not currently sniffed, while the other boxes are labeled with the name of the sniffed agent. Each labeled arrow represents a message sent from the agent at the tail of the arrow to the agent at the point of the arrow.
To view the details of a message, right-click on the arrow representing this message then click the View Message box that appears. After clicking View Message, a window appears, as shown in Figure 4.5: it lists the content, the receiver, the sender, the ontology, the language of the message, and other information.

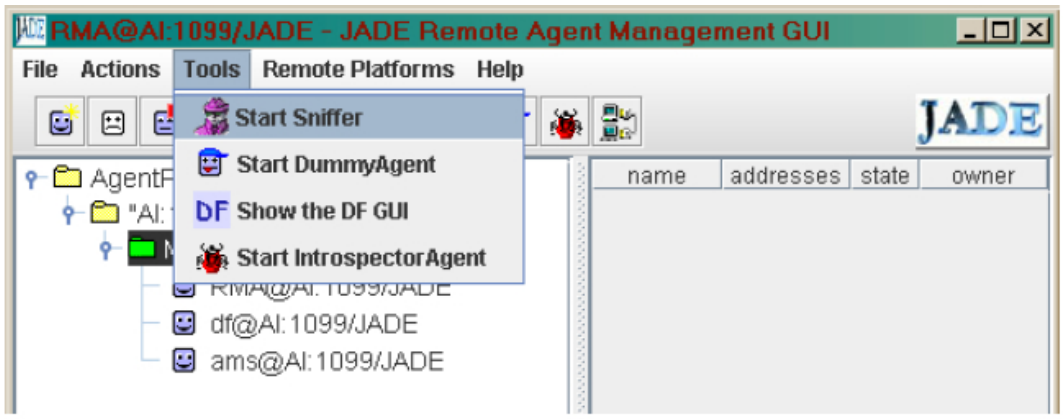

Figure 4.3: Starting the sniffer agent.

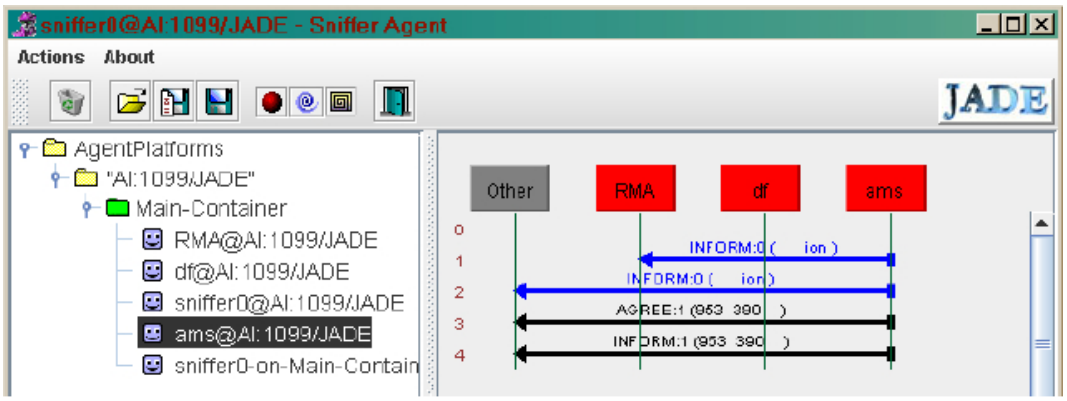

Figure 4.4: Sniffer agent's GUI.

The Sniffer Agent not only allows one to view every message, but also to save it (or all of them) to a file that can be reloaded in this GUI at a later time.
Another interesting agent that is useful in debugging the MAS is the Introspector Agent. This agent has a GUI that, as the name suggests, allows a sort of introspection of the single agents.
To start this agent from the RMA GUI, the user can either press the appropriate button or click on the Tools menu and then select the Start Introspector Agent item from the displayed menu.
This agent can monitor and control the life cycle of a running agent, and also view the messages that it has sent and received. The user is able to look at both queues of messages of the agent: the Incoming Messages and the Outgoing Messages queue.
In the Incoming Messages panel are listed the Pending and Received messages,

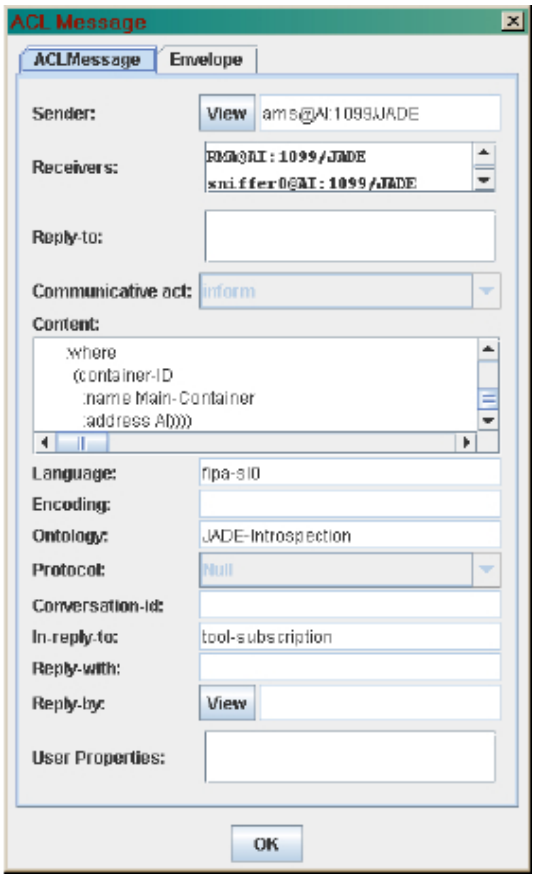

Figure 4.5: ACL message window created by the Sniffer agent.

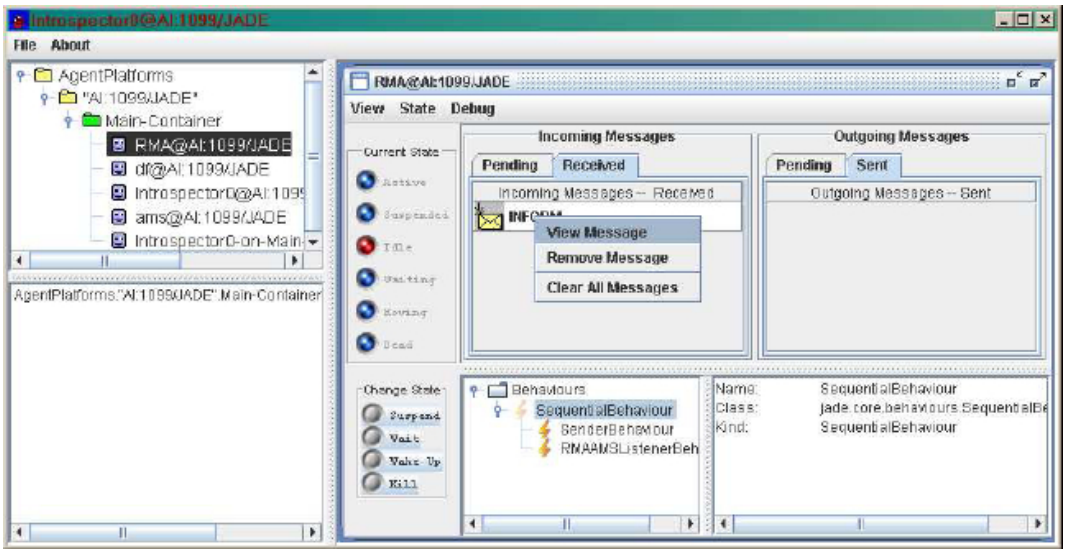

Figure 4.6: Introspector agent's GUI

labeled by their performative; by right-clicking on one message, the user can view it, remove it or remove all the messages. In a similar manner, the Outgoing Messages panel lists the Pending and Sent messages. The Introspector Agent also offers the possibility to monitor the queue of behaviors of an agent, including their execution step-by-step.

4.3 DCaseLP

DCaseLP (Distributed CaseLP) [Mig02, AMMM02], is a rapid prototyping software environment that supports the development of MASs and, as its predecessor CaseLP [MMZ99, BDM+99], has been designed and developed by the Logic Programming Group at the Department of Computer Science of the University of Genova in Italy.
DCaseLP aims at providing the developer of a MAS with an AOSE methodology and a software environment to be used during the requirements analysis, the design and the development of a working prototype. A fundamental goal is to support the development of MASs consisting of multilingual agents. More precisely, more than one language must be available not only to specify the agents belonging to the system, but also to define their architecture, behavior and state, allowing both existence and communication in the environment of agents created using such different languages.
As its name emphasizes, DCaseLP has not been created from scratch, but has come into being to overcome deficiencies present in its predecessor CaseLP: the latter supports multilingual agents at the specification level, but not at a lower level, since all the agents are coded in the logic programming language Prolog. The previous, and first, release of DCaseLP [AMMM02] overcame most of CaseLPs limitations but did not allow to develop a completely multilingual prototype. The subsequent work of integration of logical agents (described in another thesis [Gun05]) has lead to the current working version of DCaseLP, which provides an additional language to implement agents in the MAS but is not, yet, the final version since it is an ongoing work and can be subject to additional integrations.

4.3.1 DCaseLPs focuses

It is quite clear that the development of a working prototype of a MAS can require a long time and different skills, and is more complicated if the system is composed of heterogeneous agents. During prototyping, the heterogeneity of agents emerges from three different features:

- the specification language;
- the architecture;
- the implementation language.

Specification language: during the specification stage, the developer might need, maybe for only some of the agents, to emphasize particular features of their behav-

ior and, to do so, would like to use one or more different specification languages than the one currently in use and probably satisfactory for specifying other agents in the system.

Architecture: it determines the way the agent reasons, represents its knowledge, plans its behavior and makes decisions. The developer must be able to choose from the variety of architectures that have been defined in the agent community, in order to maintain the differences existing between the agents being modeled.

Implementation language: the developer should be able to define the behavior and the state of each agent using the more appropriate language, taking into account the architecture of the agent itself or the particular application domain, instead of being forced to implement all the agents with the same language. By integrating into the same running prototype different implementation languages, some of which directly executable, it is also possible to directly implement some of the agents, skipping the specification stage when it is not relevant.

A software environment claiming to integrate heterogeneous agents must show the basic attribute of multilingualism in all these three aspects, and this is what DCaseLP wants to accomplish.

The development strategy that DCaseLP proposes (refer to [Mig02] for details) is specifying each view of the MAS as, for example, the MAS architecture, the interaction protocols between agents, the internal architecture and functioning of each agent using the language that is most suitable for the current description, and to subsequently verify, execute, or animate the obtained specifications. These specifications are checked not through formal validation and verification methods (not yet dealt with in DCaseLP) but by producing an executable code and running the developed prototype. It should be clear by now that the methodology proposed by DCaseLP, together with the feature of providing more than one language to define different aspects of an agent, allow to consider it an instance of the ARPEGGIO framework ([DKM+99]), as its predecessor CaseLP.

Agents need to interact and exchange information in order to cooperate or compete for the control of shared resources; this interaction may follow sophisticated communication protocols to which the developed prototype must adhere.

Such interaction protocols are considered a basic matter in DCaseLP and, as detailed further in this chapter, DCaseLP provides the developer with the means to exactly specify which agents can take part to a conversation and in which order they interact, making the debugging of the prototype much easier. Before describing DCaseLP, it is best to detail the features characterizing CaseLP, so the reader will have a better understanding of the DCaseLP environment and of the overall ideas behind it that have brought it into existence.

4.3.2 CaseLP

CaseLP [MMZ99, BDM+99] provides a well-defined prototyping method, as well as a set of tools and languages which support the developer during the realization of the MAS prototype and are helpful in testing distributed software applications.
A logic programming language has been chosen as the basis for the prototyping environment because it is powerful, declarative and an executable specification language and, together with the agent technology, can play a very effective role in the rapid prototyping, testing and refinement of a wide spectrum of software applications.
At the *system specification level*, an architectural description language can be used to describe the prototype in terms of agents classes, their instances, the services they provide/require and their communication links.
At the *agent specification level*, a rule-based,not executable language can be adopted to easily define reactive and proactive agents. An executable, linear logic language can define more sophisticated agents and the system in which they operate. Furthermore, an imperative language, HEMASL [Mar99], can describe both the agents architectures, the agents classes, their instances and the environment in which they are embedded.
Finally, at the *implementation level*, a Prolog-like language, extended with additional primitives, has been used. Obviously, in order to really be a supporting development environment, CaseLP also offers simulation tools to visualize the execution of the prototype and to collect statistics about it. In Figure 4.7 the reader can see the main languages that CaseLP offers to the developers.
The communication among agents takes place through message passing and the agent communication language used is KQML [LM95] based on speech-acts, although any other communication language can be easily adopted by the agents.

The Limitations

By providing a set of specification languages for the definition of the behavior of the agents, as well as a methodology to translate/integrate them into a unique executable specification, CaseLP has succeeded in obtaining multilingualism in both the specification and the architecture of the agents, though missing the multilingualism in the implementation language. Besides this lack, CaseLP also has other limitations:

1. centralization;
2. poor support to concurrency;
3. limited portability.

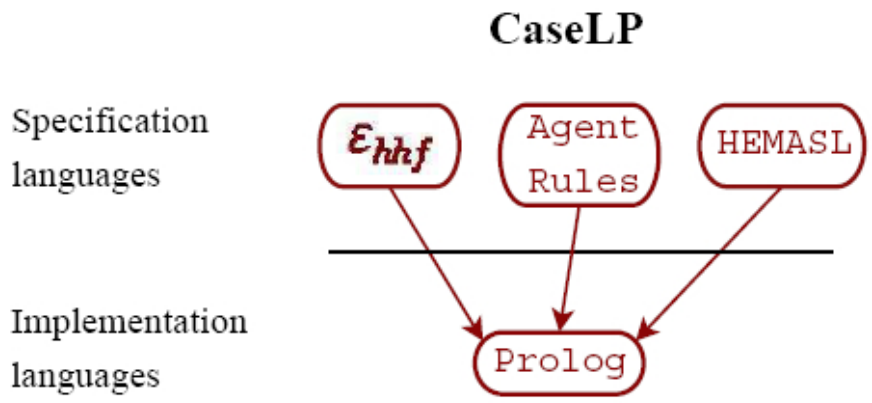

Figure 4.7: The main languages available in the CaseLP environment.

Centralization In CaseLP, the prototype is executed through a centralized round-robin scheduler that activates all the agents in turn in the MAS, following a fixed cyclic order. Once an agent is activated, it behaves accordingly to the rules that define the actions it should take, and when it has terminated the scheduler activates the next agent. The scheduler controls the global clock of the MAS by managing the simulation time, and also handles the exchange of messages between agents.
Concurrency CaseLP does not allow the real concurrent execution of agents, since there is no way to have more than one agent activated in the MAS at the same time. As we have said above, the CaseLPs scheduler activates only one agent at a time, and it is not possible to have more than one scheduler running in the MAS: concurrency among agents is only simulated.
Portability CaseLP was initially developed as an extension of a constraint logic programming language with theories, communication predicates and 'safe' state update predicates (that guarantee no permanent effect in case of failure), therefore it has been implemented in Prolog. Unfortunately, at present Prolog is not widely used as a programming language for (commercial) applications, thus there is not a Prolog interpreter to consider 'portable' to the systems available in the industry domain.
These features are, nowadays, a must for the majority of commercial (and not) applications, thus they had to be added in some way. Since it did not seem easy to achieve such characteristics directly from the Prolog infrastructure on which CaseLP is built upon, the adopted solution has been to develop a totally new environment, DCaseLP indeed, based on another programming language (Java™[Sun]) widely used nowadays, with the intention of realizing the same project from which CaseLP originated.

4.3.3 DCaseLPs first release

The first version of DCaseLP has been developed mainly concentrating on overcoming the three above-cited lacks: such purpose has been fulfilled through the introduction of JADE™(Java Agent DEvelopment Framework [Til]),a software framework fully implemented in the Java programming language and whose minimal system requirement is the version 1.4 of the Java run-time environment.
Since JADE runs as a Java application, it runs in a JVM (Java Virtual Machine) and, therefore, is portable on most of the available operating systems. Agents in JADE are implemented with Java threads, thus it is possible to execute more than one agent simultaneously, achieving the aimed concurrency. Distribution comes directly from the possibility of distributing JVMs and from Javas *Remote Method Invocation* (RMI) mechanism.
This release of DCaseLP represented a step forward towards realizing the aimed multilingualism since it proposed two implementation languages: the Java language and the Jess [FH] expert system language. Unfortunately, the multilingualism of this version was still too limited and CaseLPs languages/tools were quite far from being exploitable by it.
As far as the multilingualism at the specification level is concerned, this release offered the possibility to specify some aspects of the MAS using AUML (Agent-based Unified Modelling Language) [FIP], and automatically create Jess agents from the AUML specification.
In Figure 4.8, we have shown the languages available in the DCaseLP environment.

The new agents: Java and Jess The introduction of the JADE platform has, at first, lead to the new type of agents executable in a prototype of MAS: the Java agents.
JADE is a framework for implementing working MASs and provides many functionalities that facilitate the management of all the agents that constitute the system. As it is, it cannot be considered a real software development tool for MASs because it does not support the entire software engineering process: it does not provide any means by which to create an abstract model of the system to develop, nor by which to easily pass from the designed components/agents to the software objects.
It supplies many Java packages among which there are the ones containing the classes to use to implement agents, their ontologies and their behaviors. An agent is created extending an appropriate class that allows to take basic actions and to define new ones (more details can be found in JADEs documentation downloadable from http://jade.tilab.com/). Both the knowledge and the behavior of these agents are entirely implemented in Java, and this is why they are called Java agents.
Beside the Java agents, another type of agents was introduced in the first version of

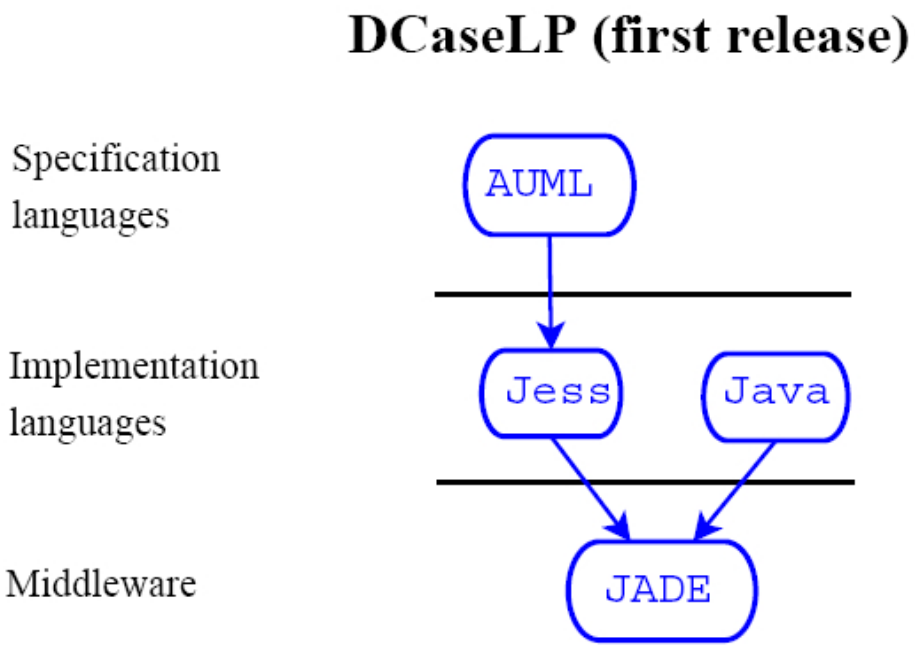

Figure 4.8: The languages available in the first release of the DCaseLP environment.

DCaseLP: the Jess agents.
Their behavior and knowledge is represented using declarative rules written in Jess, a rule engine and scripting language entirely written in Java. Jess is a language inspired by the CLIPS [NAS] expert system shell: it allows to create a knowledge base and to interact with it through the use of an inference engine; in other words, it adds to applications the capacity to reason using knowledge supplied in the form of declarative rules.
A Jess agent is a JADE agent extended in order to embed a Jess interpreter and Jess rules. The architecture is a deliberative one: it is composed of a list of Jess rules defining the behavior of the agent, a list of Jess facts representing its internal state and a Jess interpreter that represents the core of the agent.
Once the rules representing the behavior of an agent have been specified, it is possible to create a Jess agent embedding such rules and then run it in the JADE platform (more details can be found in [AMMM02]).
By using DCaseLP, therefore, the developer can create in a semi-automatic way agents that satisfy the interaction protocols and the UML/AUML diagrams and, by executing the prototype, can check if the requirements analysis has been correctly carried out.
The idea of translating UML and AUML diagrams into a formalism and checking their properties by either animating or formally verifying the resulting code is shared by many researchers working in the AOSE field [Hug02, SA03]. The reader interested in more detail on the passages that lead to the Jess agents (starting from the diagrams) is referred to [AMMM02].
The semi-automatic translation from UML/AUML diagrams into Jess agents is achieved by exploiting two different configuration files that are written in the XSL format, which is normally exploited to express style sheets. These configuration files have been developed as part of the work of the master thesis described in [Mig02].
The Limitations Two are the missing features of the first release of DCaseLP on which we have focused our attention on:

- the possibility to reuse the Prolog-based code and instruments already developed for CaseLP;
- the ability to reason about properties of the interactions occurring between the agents.

A step forward towards recuperating the functionalities offered by CaseLP has been done with the integration of tuProlog into DCaseLP [Gun05].
CaseLP is implemented in SICStus Prolog [SIC] and its agents are mainly SICStus Prolog code extended with adhoc communication primitives. A lot of work has

been done to study and define semi-automatic translators from high-level specification languages to CaseLPs implementation language: the environment contains tools that semi-automatically translate c [Del97], HEMASL [Mar99, MMMZ00] and AgentRules [Mas02] into Prolog.
$\mathcal{E}_{\langle\langle\{}$ has been, for example, used to model high-level interaction protocols in applications developed with CaseLP.

4.3.4 DCaseLPs current release

In the current release of DCaseLP there are now kinds of agent that can be run in a developed prototype:

- Java agents
- Jess agents
- tuProlog agents

tuProlog [DOR01] is a declarative logical language for which there exists an inference engine written in Java. The tuProlog agents are logical agents intended as logic programs: the behavior of these agents (as well as their knowledge) is, in fact, implemented by a Prolog theory.
The next step to take to achieve the aim is to recuperate (from CaseLP) the use of $\mathcal{E}_{\langle\langle\{}$, AgentRules and HEMASL: we think that having added Prolog to DCaseLP, this task will be more easy to accomplish.
Verification of specifications. By executing the $\mathcal{E}_{\langle\langle\{}$ specification of the MAS, the prototype can be tested and its correctness verified by using the $\mathcal{E}_{\langle\langle\{}$ interpreter. CaseLP is, thus, capable of providing a limited support to formal verification of specifications.
Without introducing the Prolog language, all that work would have not been exploitable from DCaseLP. The ability to specify agents as Prolog theories should make it easier to use the tools available in CaseLP, since the latter is totally implemented in a logic programming language.
Reasoning about interaction protocols means to check if a certain set of properties holds after a conversation has taken place. This can, for instance, allow to determine which protocol (from a set of available ones) satisfies a goal of interest, or it can help to find out which protocols can be combined to accomplish complex tasks.
After proving desired properties of the interaction protocols, the developer can animate them through the facilities offered by DCaseLP.

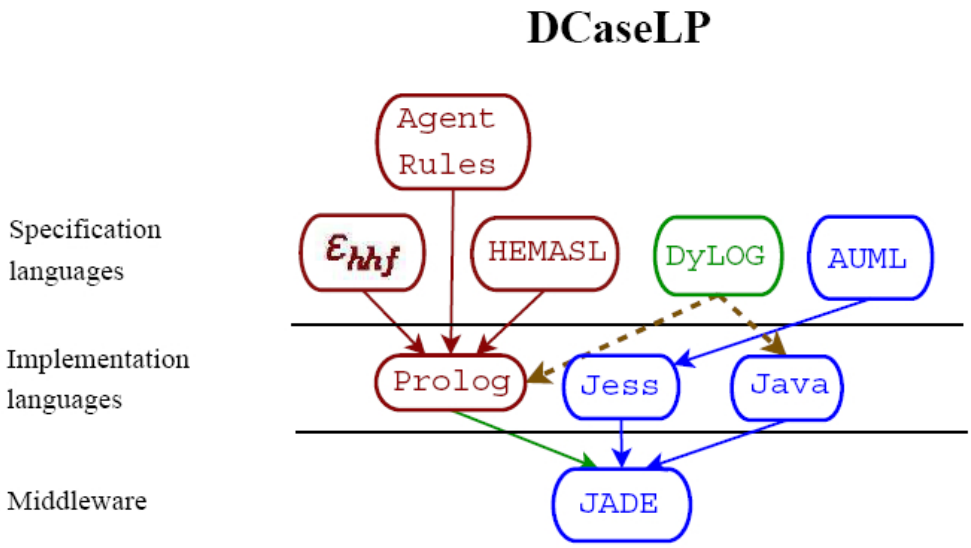

Figure 4.9: The languages available in the current and future release of the DCaseLP environment: the dotted lines are part of the future work.

Chapter 5

Analysis and design of auction mechanisms

5.1 Introduction

As stated in the previous chapters, auctions are procedures for allocating goods that rely on competitiveness and lack of information on the actual market price of the goods. The main reasons for using auctions are to extract an higher surplus (from the Auctioneer's point of view) and to pay less for the goods (from the bidders' point of view). Hence, a careful choice of the auction mechanism to employ and the capability of foreseeing the outcomes of the auction are necessary to obtain the desired results.

Considering that the Dutch auction mechanisms is completely equivalent under any value model to the first-price sealed-bid auction, we have implemented the three standard mechanisms described in Chapter 3: English, first-price sealed-bid and second-price sealed-bid mechanism. For the English auction mechanism, two different versions have been implemented: one with continuous bidding and the other that make use of bidding rounds.

Each auction mechanisms require two types of agent at least:

1. The *Auctioneer* agent that puts items on sales, receives offers, distributes information on what is going on and decides the auction winner

2. The *Bidder* agents that try to buy the items on sale by evaluating newly acquired information and sending offers.

In the next sections we are going to analyze one by one the different auction mechanisms implemented, detailing:

- the communication protocols, using a specialized form of UML Interaction diagrams,

- the design of the implemented agents, using a Pascal-like pseudo code.

In the next chapter we will describe the outcomes of our test, obtained from running our agents in different value models and we will compare the empirical results with the theoretical ones.

5.2 Common Features

Before an auction can take place, the potential buyers' agents must be aware that someone is selling some object, they must have at least a minimal interest in buying the object on sale and they must also know when and where the auction is going to take place: these aspects could be resolved introducing 'searchers', agents that wander around the network to gather information on the scheduled auctions (see [Mae94]), and bulletin boards where auctioneers can advertise the auction that they are preparing. However, we will not investigate these aspects any further because outside of the scope of this thesis. Instead we will concentrate only on the mechanisms of offering and evaluating bids.

There are three main phases in every implemented auction mechanism:

1. registration phase

2. bidding phase

3. object attribution phase

Registration phase The registration phase has been introduced because, given that all bidders know who is the auctioneer and what is selling, it is necessary to the auctioneer to be aware of who are the interested potential buyers; thus, all the implemented auction mechanisms have a first phase in which each potential buyer registers himself as a bidder in the specific auction, declaring that they want to receive all the information about the starting auction.

This phase is implemented in this way: as soon as the bidder comes to know that an interesting auction is going to take place, it sends a message to the auctioneer where he asks to be registered, and then waits for a confirming message; when the message arrives, the bidder knows that is officially participating to the auction and can start to think about his strategy. The auctioneer could decide not to allow the bidder to enter the auction, and this could happen because of a negative evaluation given by some reputation system (e.g. based on behaviors during past auctions as seen in [CH04]) but we will not handle this case.

Offering phase There are two common features in the bidding phase (that, otherwise, differs from mechanism to mechanism): auction duration and reserve price. In all the mechanisms that we implemented, the auctioneer determines what is the bidding deadline, and communicates it to the registered bidders: this permits them to use also time-based strategies (supposing that all the agents have synchronized clocks). Also the reservation price is an information that the auctioneer distributes to all the bidders and is an implicit warning that no offer lesser than the reservation price will be accepted: this feature forces the bidders to offer a price that is already acceptable to the auctioneer. The reservation price can be set to zero if the auctioneer has no minimal estimate of the object on sale.

Object attribution phase An aspect of the third phase that is common to all the auction mechanisms (except for the English auction mechanism with continuous bidding) is that the highest bid could be offered by more than one bidder simultaneously. If this happens, it is necessary to decide which bidder gets the object on sale: in our implementation we assign the object using a lottery with a uniform probability distribution because there are no information that permit to distinguish one bidder from the other; if a reputation system was used, we could use a lottery weighed on the reputation of the bidders to give more probability of winning to the bidder considered more reliable.

In every mechanism, when the auction time is over and the winner has been selected, it is important for the auctioneer to receive a confirmation from the winner

in which he agrees to pay for the object on sale: in our implementation, the auctioneer waits a determined amount of time for the confirmation message and, if it does not arrive, it can restart the auction. If a reputation system was implemented, the auctioneer could send a negative feedback to the agents that maintain the database of bidders' reputation. If the confirmation message is sent from the winner to the auctioneer, the auction terminates.

5.3 Communication Protocols

The Unified Modeling Language (UML) is gaining wide acceptance for the representation of engineering artifacts in object-oriented software. The view of agents as the next step beyond objects leads to explore extensions to UML and idioms within UML to accommodate the distinctive requirements of agents.
To pursue this objective, recently a cooperation has been established between the Foundation of Intelligent Physical Agents (FIPA) and the Object Management Group (OMG). As a first result of this cooperation, Bauer, Muller and Odell proposed a framework of AGENT UML (AUML) ([BMO01]) where they introduced, among other ideas, a new class of diagrams: *interaction protocol diagrams*.
Interaction Protocol (IP) diagrams extend UML state and sequence diagrams in various ways. Particular extensions in this context include agent roles, multithreaded lifelines, extended message semantics, parameterized nested protocols, and protocol templates.
In the first stage of analysis we described the communication protocols of the implemented auction mechanisms. We opted for AUML IP diagrams because they suited our purposes very well: in fact, an agent IP describes

- a communication pattern, made of an allowed sequence of messages (between agents having different roles) and constraints on the content of these messages

- a semantics that is consistent with the communicative acts (CAs) within the communication pattern.

We introduced the communicative acts in Section 4.2.2. Given the fact that the auction mechanism agents run on the JADE platform (described in Section 4.2) and that JADE adheres to the FIPA standards, it seemed natural to take advantage of these type of diagrams to model the exchange of messages between the auctioneer and various bidders.

5.3.1 Sealed-bid auction mechanism

In our library, we decided to implement classic first-price and second-price sealed-bid auction mechanisms for a single indivisible object, as described in Chapter 3. Analyzing these mechanisms we realized that they share the same communication pattern, so we produced a single interaction protocol, shown in Figure 5.1.
Let us describe this protocol. The Auctioneer agent waits for `register` messages denoted by a `request` communicative act by which agents declare their will to enter the auction as Bidders and to receive all the information about the proceeding of the auction.
The Auctioneer could reply with a `confirm` communicative act with a message that informs the Bidder agent of its actual registration, or with a `refuse` act, denying to the agent the participation to the bidding phase.
After the registration phase has finished, the Auctioneer agent sends a message with an `inform` communicative act where it stores all the information about the auction: what is the reservation price, when the auction finishes and how many bidders are participating. At this point the Auctioneer waits for Bidders' offers and stores all the them until the auction time expires. The participating agents could also submit no bid. At the end of the auction, the Auctioneer examines all the offers and decides what is the best one, then sends an `inform` message to all the bidders with the highest bid to notify them that they can win the auction and how many other agents are in the same situation. Hence, the Auctioneer determines the real winner and broadcasts an `inform` message to declare who got the object on sale and at what price. Last, the auctioneer waits for a `confirm` message from the winning Bidder agent.

5.3.2 English auction mechanisms with continuous bidding

We implemented an English auction mechanism for a single indivisible object as described in Chapter 3. Our analysis of this auction led us to the definition of the interaction protocol in Figure 5.2. Let us see the communication protocol in detail.
In the registration phase, p Bidder agents ask to be registered in the Auction by sending a message with communicative act `request`, the Auctioneer can accept the request (sending back a message of `confirm` to each accepted agent) or deny the request (with a `refuse` communicative act).
Once the registration time is over, the Auctioneer sends an `inform` message to the n registered agents specifying its reservation price, this warns the Bidder about the minimal acceptable offer. Then the Auctioneer sends another `inform` communicative act to start the offering phase.

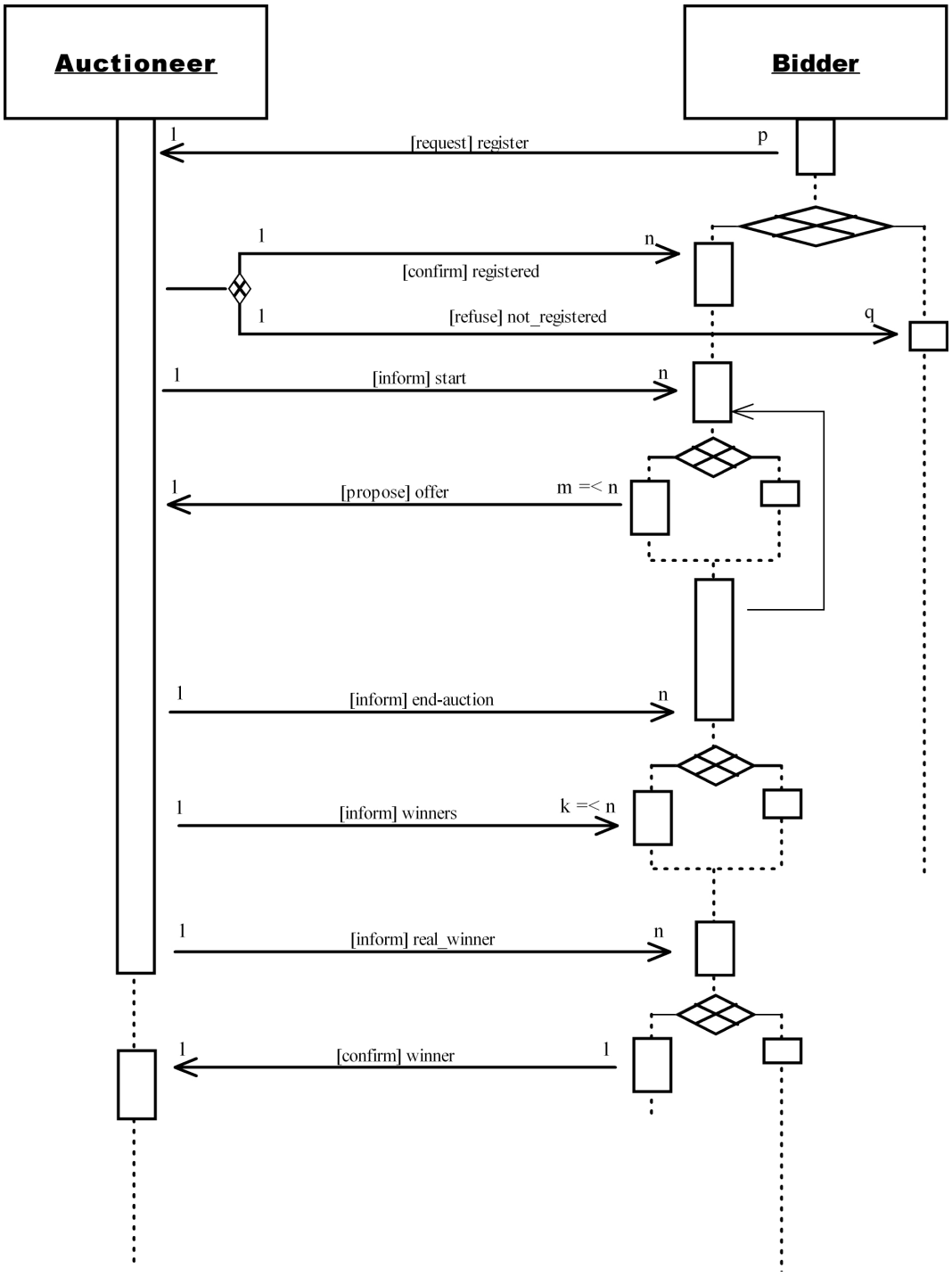

Figure 5.1: Sealed-bid auction mechanism

In the offering phase, the Bidder agents send `propose` messages that contain offers: every time a Bidder x offers a bid that is better than the highest received bid, the Auctioneer sends an `inform` message back to x to notify that is winning the object Then the Auctioneer has three possibilities:

1. broadcast to all n participants what is the new highest offer
2. broadcast to all n participants that there is an extension to the original auction span
3. declare the end of the offering phase

All these possibilities are communicated by `inform` messages and each of them causes different behaviors of the Bidder agents: the first two messages leave to the Bidders the chance to make new offers (shown in the Figure 5.2 by the loop back arrows), while the last message moves the communication protocol to the next phase, the object attribution.
The object attribution phase of an English auction mechanisms with continuous bidding is simple because the evaluation of the best bid is completely done in the offering phase, so the winner agent is already determined once that phase is finished. Thus, the Auctioneer broadcast an `inform` message with the name of the winner, then wait for a `confirm` message.

5.3.3 English auction mechanism with rounds

The communication protocol of an English auction mechanism implemented with rounds is almost identical to the continuous version, except for the object attribution phase. In fact, thanks to the mechanism of rounds, all the best offers are taken into consideration, so at the end of the offering phase, more than one Bidder agent could claim the object on sale. Thus, the Auctioneer sends an `inform` communicative act to the possible winning agents that contains their number, then, after determining the real winner, sends another `inform` message declaring who is the winner. We can see this mechanism in the interaction protocol of Figure 5.3.

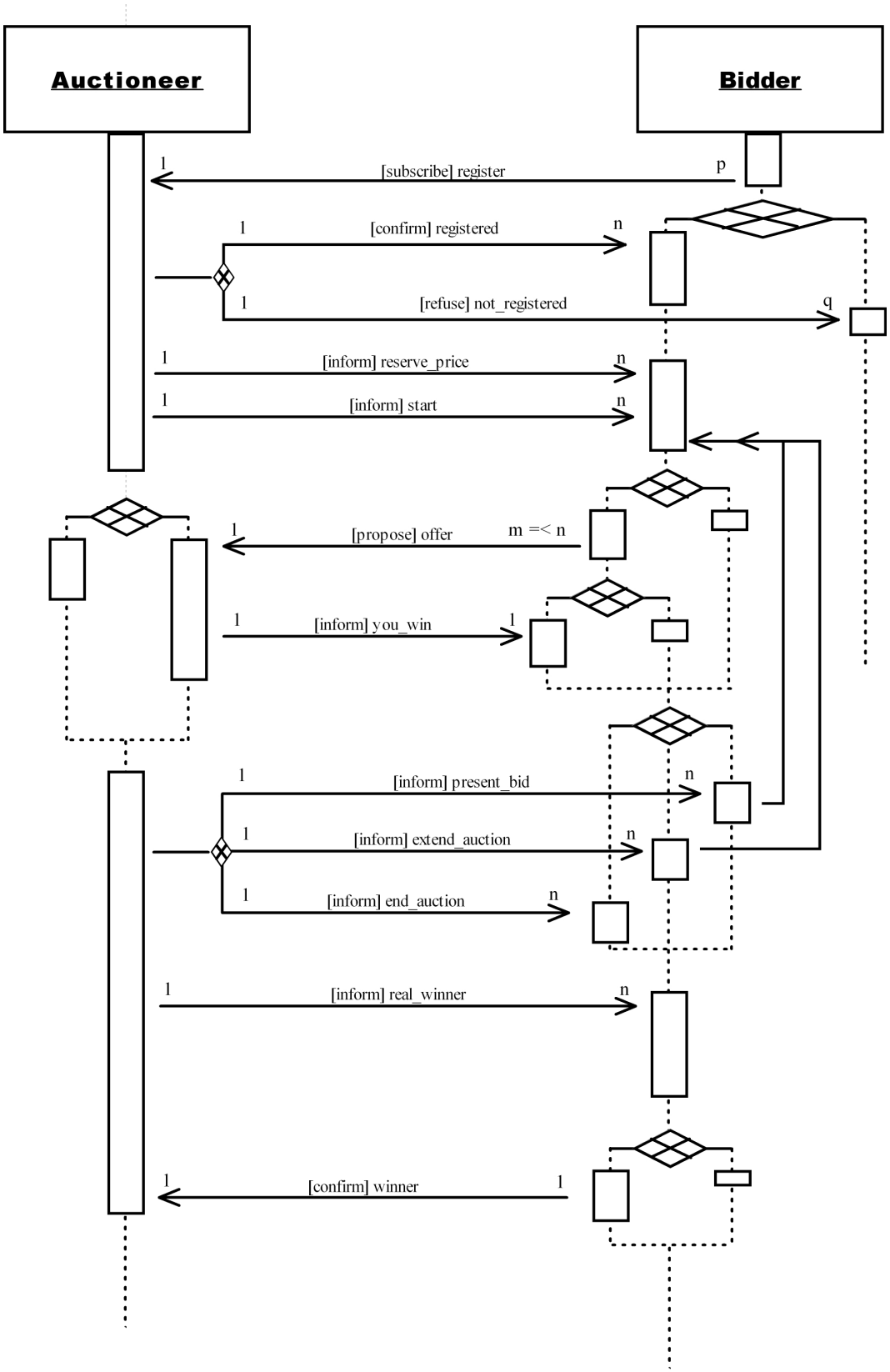

Figure 5.2: English auction mechanism with continuous bidding

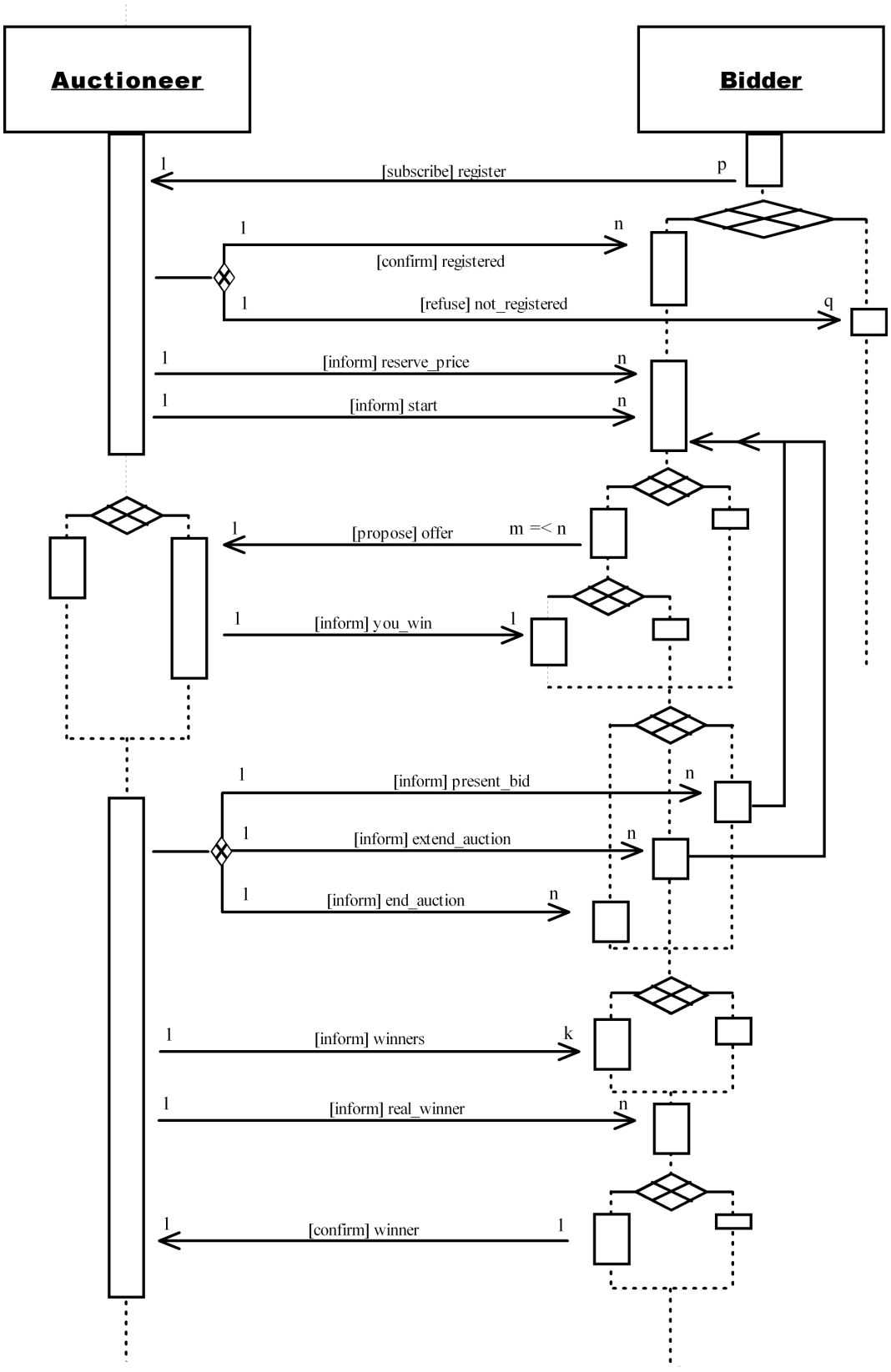

Figure 5.3: English auction mechanism with rounds

5.4 Design of auction mechanisms

The next step in our analysis was to describe the behavior of Auctioneers and Bidders of each auction mechanism, and we decided that Pascal pseudo-code was the right tool for this activity.

5.4.1 Registration phase

The registration phase is common to all mechanisms, so it is reasonable to treat it on its own. In our implementation, the Auctioneer agent sets a Registration-span variable to some value that indicates how long this phase will last in minutes, then it calculates the end of the Registration phase and puts it in the `End-registration` variable: we suppose that the Auctioneer has made public this information such that any agent could use it to register. We suppose also that the function `now()` returns the present time.

Listing 5.1: Auctioneer: Registration phase

```
1  Registration-span := ...;
   Registered-list := empty-list;
3  End-registration := now() + Registration-span;
   while (now() < End-registration)
5  do
     if (receive('request', "Register", Sender))
7    then
         if accept(Sender)
9        then
             add Sender to Registered-list;
11           send('confirm', "Registered", Sender);
         else
13           send('refuse', "Failed_Registration", Sender);
         endif
15 endw
```

The Auctioneer waits until End-registration, looking for `request` messages that asks for registration; then it discriminates between acceptable and not acceptable Bidder agents, and adds the first ones to the Registered-list. As we said before, the `accept()` that varies from auctioneer to auctioneer and we do not define here. On the other hand (Listing 5.2), the Bidder agent always tries to register, then waits for a message from the Auctioneer: if a message of confirmation arrives, it participates to the auction, otherwise it can get either a `refuse` message or no message at all,

in such case it stops interacting with this Auctioneer agent and look for another auction.

Listing 5.2: Bidder: Registration phase

```
Registered := false;

send('request, "register", auctioneer);
while (now() < Registration-time)
do
  if (receive('confirm', "Registered", auctioneer))
  then
      Registered := true;
  elseif (receive('refuse',"Failed_Registration",auctioneer))
  then
      look for another auction;
  endif
endw
```

5.4.2 Sealed-bid auction mechanisms

Both implemented sealed-bid mechanisms (first-price and second-price) share the same design. After the registration phase, the Auctioneer agent calculates the time when the offering phase should finish (`End-Auction`) and then sends a message to all registered Bidders to inform that the auction has started, that `Bidder-Number` Bidders are participating and that the minimal acceptable offer is `reservation-price` (Listing 5.3). The information on the number of participants is useful to each Bidder to determine its (possibly optimal) strategy.

Listing 5.3: Sealed-bid Auctioneer : Offering phase

```

Auction-span := ...;
Reserve-price := ...;
Bid-list := empty-list;
End-auction := now() + Auction-span;
Bidder-number := length(Registered-list);
multicast('inform',
 "start(Reserve-price, End-auction, Bidder-number)",
 Registered_list);

while (now() < End-auction)
do
```

```
28   if receive('propose', "offer(New_bid)", Sender)
     then
30       add (New-bid, Sender) to Bid-list;
     endif
32 endw
   multicast('inform', "End-auction", Registered_list);
34
   (Winning-bid, To-pay-bid, Winner-list) :=
36                   eval-offer(Bid-list, Reserve-price);
```

Then the Auctioneer loops until the auction time expires, storing all the offers received in a list (`Bid-list` line 30). When the auction time finishes, the Auctioneer evaluates all the offers contained in the `Bid-list` and chooses the best ones (lines 35-36), determining the bidders, the offered price and the price they would have to pay if they really win the auction. In the second-price sealed-bid auction the price to pay is the second best offer, while in the first-price sealed-bid auction the price to pay is equal to the offered price.

Listing 5.4: Sealed-bid Auctioneer : object attribution phase

```
38 if (length(Winner-list) == 0)
   then
40     Winner := null;
   elseif (length(Winner-list) == 1)
42 then
       Winner := Winner-list[0];
44 else
       multicast('inform',
46      "winners(Winning-bid, To-pay-bid, Winner-list)",
         Winner-list);
48     Winner := lottery-extraction(Winner-list);

50 endif
   multicast('inform', "real_winner(Winner)", Registered_list);
52
   if (Winner == null)
54 then
         restart auction with a different Reserve-price
56 else
         Wait-confirm := now() + Confirm-span;
```

```
58          Confirmed := false;
            while (now() < Wait-confirm)
60          do
              receive('confirm',"winner-confirm(Signature)",Winner)
62                  if (verify(Signature, Winner))
              then
64                Confirmed := true;
              endif
66          endw

68          if Confirmed
            then
70              auction ends correctly
            else
72              restart auction
            endif
74  endif
```

Next, as we can see in Listing 5.4 the Auctioneer agent counts how many Bidder agents offered the best bid:

- if no offer was made, the `Winner` variable is set to null
- if only one Bidder agent made the best offer, the `Winner` variable is set to its name
- if more than one Bidder made an equivalent best offer, the Auctioneer has to determine the winner with a lottery (see line 48).

The lottery could be implemented in many ways, giving more probability of winning to the agents with better reputations, for example.
After informing every registered agent on the identity of the winner, the Auctioneer waits for a message of confirmation from the winner, obviously only if the `Winner` variable is different from null. We believe that this confirmation should be a formal commitment and, as such, should be signed with a cryptographic system, PGP for example [ASZ91].
Listing 5.5 shows the pseudo code for a Bidder agent.

Listing 5.5: Bidder: Offering phase

```
define Value-model;
16  Offering_chance := false;
```

```
18 if (not Registered)
   then
20   look for another auction;
   else
22 while (true)
     switch receive(Message)
24     case ('inform',
            "start(Reserve-price, Bidder-number, End-auction",
26                                                  auctioneer)
           Offering_chance := true;
28
       case ('inform', "end-auction", auctioneer)
30         Offering_chance := false;

32     case ('inform', "winners(WinBid,WinList)", auctioneer)
           update-value-model(WinBid, WinList, Value-model);
34
       case ('inform', "real_winner(Winner)", auctioneer)
36         update-value-model(WinBid, [Winner], Value-model);
           Bidder-name := get_my_name();
38         if (Winner == Bidder-name)
           then
40             create_signature(Signature);
               send('confirm',
42                 "winner-confirm(Signature)", auctioneer);
           endif
44         exit while;

46     else
           if (Offering-chance)
48         then
               New-bid := eval-offer(Reserve-price, Value-model,
50                                    Bidder-number, End-auction);
               if (better(New-bid, Reserve-price)
52             then
                   send('propose', "offer(New-bid)", auctioneer);
54                 Offering-chance := false;
               endif
56   endc
   endw
58 endif
```

The first thing that a Bidder does is to define its value model: this can be a private-value model or a common-value model, as we explained in Chapter 3. In the first case, the Bidder defines directly the worth of the object, while in the second case the function to evaluate the object is common knowledge and the Bidder calculates this function using his own information (that can be incomplete or inaccurate). The value model permits to the Bidder to evaluate the object's price, therefore to evaluate the offer to bid.
If the Bidder is registered then it enters a loop where it processes the received messages and reacts accordingly. In a sealed-bid auction, the Bidder has only one chance of offering: this chance is given at every loop with a call to the function `eval-offer`, it is enabled when the Bidder receives the `start` message and disabled when it receives `end-auction`. Once offered, there is no way of retouching the offer, as we see in line 54. The function `eval-offer` depends strictly from the `Value-Model` which in turn depends on the knowledge of the Bidder and this knowledge depends on experience too: once the Bidder receives a `winners` or a `real_winner` message, it could update its value model to reflect what happened in the auction.
When the Bidder receives a `real_winner`, it compares its name with the winner's name: if it matches, it sends a signed confirmation message to the Auctioneer. Then, the Bidder exits the loop.

5.4.3 English auction mechanisms with continuous bidding

After the registration phase, common to all the mechanisms (see Subsection 5.4.1), the Auctioneer calculates all the timings: start of the auction, end of the auction and alarm. While the first two have the same meaning as in the sealed-bid auctions, the alarm is a new feature: we specify a period of time, specifically from the alarm to the end of the auction (line 20 in Listing 5.6), in which any new received offer will cause an extension of the auction time. The reason for the introduction of this feature is that, if a Bidder makes an offer in a period near the end of the auction, maybe the question of the object's price is still not settled: thus we set another period of time (whose value is contained in the `Extension-span` variable) in which the Bidders can submit new bids (see line 37 in Listing 5.6).
The Auctioneer then starts the auction sending a message to all the registered Bidders, containing how many of them are participating, when the auction ends and the value of the reservation price. Then the Auctioneer sets the `Present-bid` variable to the reservation price: this variable will be used throughout all the auction to

represent the last highest bid offered by the Bidders.
The Auctioneer then proceeds to the real offering phase: it loops until the end of the auction, receiving offer messages and evaluating them. If an offer is better than the last highest offer, then the Auctioneer informs the winner (line 44 in Listing 5.6), assigns the new bid to the present highest bid and warns all the other Bidders that the highest bid has changed (line 46). Moreover, if the offer arrives after the alarm time, the Auctioneer extends the auction time and warns all the Bidders of this event by sending a message with the new end of the auction (line 36-41).

Listing 5.6: English auction with continuous bidding: Auctioneer's offering phase

```
Auction-span := ...;
Alarm-span := ...;
Extension-span := ...;
Reserve-price := ...;

Start-time := now();
End-auction := Start-time + Auction-span;
Alarm-time := End-auction - Alarm-span;

Bidder-number := length(Registered-list);
multicast('inform',
          "start(Reserve_Price,Bidder-number,End-auction)",
           Registered_list);
multicast("present-bid(Reserve-price)", Registered_list);
Present-bid = Reserve-Price;
Extended := false;
Winner := Null;

while (now() < End-auction)
do
  if (receive('propose',"offer(New-bid)", Sender) and
      better-bid(New-bid,Present-bid))
  then
      if (now() > Alarm-time and not Extended)
      then
          End-auction := End-auction + Extension-span;
          multicast('inform',"auction-extend(End-auction)",
                    Registered_list);
          Extended := true;
      endif
      Winner := Sender;
```

```
44        send ('inform', "you-win", Winner);
          Present-bid := New-Bid;
46        multicast ('inform', "present-bid(New-bid)",
                     Registered-list);
48    endif
  endw
50 multicast('inform', "real_winner(Winner)", Registered_list);
```

The last part of the Auctioneer's code is identical to the one in the sealed-bid auction mechanism (see Listing 5.4).
Listing 5.7 shows the Bidder's code. After the Registration phase, the Bidder waits for a message containing the reservation price of the auction, so it can calculate the first offer to make. Then waits for the `start` message to make the first bid. Next, it starts to loop until it gets an `end-auction` message. In the loop, the Bidder reacts in this way:

- If it receives a message that contains `extend-auction(Extended-Auction)`, then it updates the `End-Auction` variable to the new end of auction (line 31);
- if it receives a `present-bid(Bid)` message and it has not received a `you_win` message previously, then it updates the `Present-Bid` value and it evaluates and submits a new offer, if it is better than the present best bid (lines 38-48);
- if it gets a `you-win` message, then it updates the `I-win` value such that it will not offer again until the best bid changes (lines 49-54).

Each time the Bidder receives a `you-win` or a `present-bid` messages, it updates its `Value-Model` because this new information could influence the evaluation of the object's price.

Listing 5.7: English auction with continuous bidding: Bidder's offering phase

```
  blocking_receive('inform',"reserve_price(Price)",auctioneer);
19 Present-bid := Price;
  Start-bid := eval-offer(Present-bid, Value-model,
21                        Bidder-number, End-auction);

23 blocking_receive('inform',"start(Bidder-number,
                              End-auction"),auctioneer)
25 if (better(New-bid, Present-Bid)
```

```
then
    send('propose', "offer(Start-bid)", auctioneer);
endif
I-win := false;

while (not receive('inform', "end-auction", auctioneer))
do
  if (receive('inform',"auction-extend(Extended-auction)",
              auctioneer))
  then
      End-auction := Extended-time;
  endif
  if (receive('inform', "present-bid(Bid)", auctioneer)
      and not I-win)
  then
      Present-bid := Bid;
      update-value-model(Bid, [], Value-model);
      New-bid := eval-offer(Present-bid, Value-model,
                            Bidder-number, End-auction);
      if (better(New-bid, Present-Bid)
      then
          send('propose', "offer(New-bid", auctioneer);
      endif
  elseif (receive('inform',"you-win",auctioneer))
  then
      I-win := true;
      blocking-receive('inform', "present-Bid(Bid)", auctioneer);
      Present-bid := Bid;
      update-value-model(Bid, Bidder-name, Value-model);
  endif

endw
```

When the offering phase is finished, the Bidder waits for a message that indicates who gets the object, it compares its name with the winner's name and, if it matches, sends a signed confirmation message to the Auctioneer. This can be seen in Listing 5.8.

Listing 5.8: English auction with continuous bidding: Bidder's object attribution

```
phase
endw

blocking_receive('inform',"real_winner(Winner)", auctioneer);

if (Winner == Bidder-name)
then
    create_signature(Signature);
    send('confirm',
         "winner-confirm(Signature)", auctioneer);
endif
```

5.4.4 English auction mechanism with rounds

The main reason for implementing a different type of English auction mechanism is that the continuous bidding version relies on time to determine the winner. Let us suppose that two Bidders submit the same bid, and it is the best bid offered until now: only the Bidder owner of the message that arrives first will be acknowledged of the best offer by the Auctioneer. This means that, in a simulated environment, the scheduler must be fair and must activate randomly the Bidders, otherwise some of them would be given a not justifiable advantage over the others.
Instead, in the round with rounds, the Auctioneer (see Listing 5.9) accepts all the offers submitted in a predetermined interval of time (that we call `round`) and only at its end the offers are evaluated: in this way, the Auctioneer needs to store a list of possible winners that is updated each round. At the end of the auction, the Auctioneer agent makes a lottery extraction over the Bidder agents acknowledged of the best offer and chooses the winning Bidder randomly, exactly as in a sealed-bid auction.

Listing 5.9: English auction with round bidding: Auctioneer's code

```
Registration-span := ...;
Registered-list := empty-list;
End-registration := now() + Registration-span;
while (now() < End-registration)
do
  if (receive('request', "Register", Sender))
  then
      if accept(Sender)
```

```
      then
10        add Sender to Registered-list;
          send('confirm', "Registered", Sender);
12    else
          send('refuse', "Failed_Registration", Sender);
14    endif
endw
16
Auction-span := ...;
18 Alarm-span := ...;
Extended-span := ...;
20 Round-span := ...;
Reserve-price := ...;
22
Start-time := now()
24 End-auction := Start-time + Auction-span;
Alarm-time := End-auction - Alarm-span;
26
Bidder-number := length(Registered-list);
28 multicast("start(Bidder-number, End-auction)", Registered_list);
Present-bid = Reserve-Price;
30 multicast("present-bid(Reserve-price)", Registered_list);
Extended := false;
32 Winner := Null;

34 while (now() < End-auction)
do
36   Round-time = now() + Round_span;
     Last-winner-list := Winner-list;
38   Present-bid := Winning-bid;

40   while (now() < Round-time)
     do
42     if (receive("offer(New-bid)", Sender))
       then
44         if (now() > Alarm-time and not Extended)
           then
46             End-auction := End-auction + Extended-time;
               multicast("auction-extend(End-auction)",
48                       Registered_list);
               Extended := true;
```

```
50          endif
            add (New-bid, Sender) to Bid-list;
52        endif
      endw
54
      (Winning-bid, Winner-list) :=
56            eval-offer(Bid-list, Last-winner-list, Present-bid);

58    switch (length(Winner-list))
      case 0:
60            Winner = "none";

62    case 1:
            Winner = Winner-list[0];
64
      else
66        multicast("winners(Num-winners)", Winner-list);
      endsw
68
  endw
70 multicast("End-auction", Registered_list);

72 Winner = lottery-extraction(Winner-list);
   send("real_winner(Winner, Present-bid)", Registered-list);
74
   if (Winner == null)
76 then
            restart auction with a different Reserve-price
78 else
            Wait-confirm := now() + Confirm-span;
80          Confirmed := false;
            while (now() < Wait-confirm)
82          do
              receive('confirm',"winner-confirm(Signature)",Winner)
84                  if (verify(Signature, Winner))
              then
86                Confirmed := true;
              endif
88          endw

90          if Confirmed
```

```
        then
92          auction ends correctly
        else
94          restart auction
        endif
96  endif
```

The Bidder agent's code is identical to the one produced for the English auction mechanism with continuous bidding except that, at the end of the offering phase, the Bidder waits for a `winners` message that contains the best bid and the list of agents that submitted it. This information could be useful for the Bidder to be used in case of future auctions with the same type of object on sale (this information modifies the Bidder's value model, as we see in Listing 5.10 on lines 60).

Listing 5.10: English auction with continuous bidding: Bidder's code

```
  define Value-model;
2 Bidder-name  = get_my_name();

4 Registered := false;

6 send('request', "register", auctioneer);
  while (now() < Registration-time)
8 do
    if (receive('confirm', "Registered", auctioneer))
10  then
        Registered := true;
12  elseif (receive('refuse', "Failed_Registration", auctioneer))
    then
14      look for another auction;
  endif
16 endw

18 blocking_receive('inform',"reserve_price(Price)",auctioneer);
  Present-bid := Price;
20 Start-bid := eval-offer(Present-bid, Value-model,
                                Bidder-number, End-auction);
22
  blocking_receive('inform',"start(Bidder-number,
24                                End-auction"),auctioneer)
  if (better(New-bid, Present-Bid)
26 then
     send('propose', "offer(Start-bid)", auctioneer);
```

```
endif
I-win := false;

while (not receive('inform', "end-auction", auctioneer))
do
  if (receive('inform',"auction-extend(Extended-auction)",
              auctioneer))
  then
      End-auction := Extended-time;
  endif
  if (receive('inform', "present-bid(Bid)", auctioneer)
      and not I-win)
  then
      Present-bid := Bid;
      update-value-model(Bid, [], Value-model);
      New-bid := eval-offer(Present-bid, Value-model,
                            Bidder-number, End-auction);
      if (better(New-bid, Present-Bid)
      then
          send('propose', "offer(New-bid", auctioneer);
      endif
  elseif (receive('inform',"you-win",auctioneer))
  then
      I-win := true;
      blocking-receive('inform', "present-Bid(Bid)", auctioneer);
      Present-bid := Bid;
      update-value-model(Bid, Bidder-name, Value-model);
  endif

endw

blocking_receive('inform', "winners(WinBid,WinList)", auctioneer);
update-value-model(WinBid, WinList, Value-model);

blocking_receive('inform',"real_winner(Winner)", auctioneer);

if (Winner == Bidder-name)
then
    create_signature(Signature);
    send('confirm',
         "winner-confirm(Signature)", auctioneer);
```

```
endif
```

Chapter 6

Simulating auction mechanisms

6.1 Introduction

Each auction mechanism in our library is implemented as a JADE-based application compounded by five agents. The files that contain the auctioneer agents' codes are named 'AUCT_*type*.pl' while the files with the bidder agents' code are named 'G*X*_*type*.pl', where *type* refers to the auction mechanism and *X* is an integer. The code of the agents is in Appendix A.
The agents are implemented in tuProlog in the DCaseLP environment on the JADE platform, thanks to the integration of logic agents into DCaseLP [Gun05]. Therefore, to simulate an auction, it is necessary to start a JADE platform and to create all the agents with class `tuPInJADE.JShell42PCycleGui`. The auctioneer must be started first, then all the bidders can be started and the auction begins.
In this chapter we will show how to simulate auction using the implemented agents. The implemented auction mechanisms are made to sell a single indivisible object, but there are many other characteristics that can be modified to simulate different situations of sale; let us see them in details.

6.2 Customizable characteristics of the implemented mechanisms

auctioneers Registration time. It is the duration of the registration phase in minutes. It is contained by the atom `reg_span(Time)`.
Acceptance of registration. The user can customize the predicate `accept_bidder` to define the rules by which an agent can be accepted as a bidder. These rules can be private of the auctioneer or depend on an external reputation system.

Auction time. It is the duration of the offering phase in minutes. It is contained by the atom `time_span(Time)`.

Alarm time. Only in the English auctions. It is the interval of time at the end of the offering phase, during which any new offer will trigger the extension of the auction time. It is contained by the atom `alarm_span(Time)`.

Extension time. Only in the English auctions. It is the interval of time that the auctioneer adds to the auction time if any offer has arrived during the alarm time. It is contained by the atom `ext_span(Time)`.

Wait time. It is the interval of time that the auctioneer waits for a message of confirmation from the winning bidder. It is contained by `wait_span`.

reservation price. The reservation price is the lowest bid accepted by the auctioneer to sell the object. It is contained by the atom `reservation_price()` and can be modified to reflect the need of the simulation. The reservation price can be also a list of values, thus creating a multi-dimensional auction; it is necessary to modify also the bid comparison feature.

Bid comparison. The auctioneer must choose if a new bid is better than another. The user can customize this feature to reflect the preferences of the auctioneer over offers by rewriting the `better-bid` predicate. Modifying this predicate and the reservation price accordingly, the user can simulate a multi-dimensional auction.

Attribution of the object. In case the auction ends with two or more bidders owning the best offer, the auctioneer must decide who is the real winner using a lottery. This lottery can be customized rewriting the predicate `lottery`.

Bidders

Value Model. The value model of a bidder determines its object's monetary worth (see Section 3.2). The default value model implemented in our bidders is the private one: the bidder asserts a static value for the object in the atom `object_value(Value)`. The user can customize this feature, defining `object_value(Value)` as a predicate that calculates the object's worth for the bidder using both private and public information.

Strategy. The strategy of a bidder determines the value and the time of its offers. It depends mainly on the value model and on other bidder's behavior, but other aspects can be considered, like time and information from sources external to the auction. To modify a bidder's strategy, the user should modify the predicate `eval_offer`.

6.3 Testing the prototypes

Our aim in this section is to test our implementation, thus we ran all the mechanisms of our library with the same parameters to show that they satisfy the fundamental Revenue Equivalence Theorem (see Theorem 7).
These are the parameters of every test that we made:

- the reservation price is set to 50 coins;
- the duration of the auction is set to 5 minutes;
- the registration span is set to 1 minute;
- there are four different bidder agent with private independent values;
- the bidder agents have private values for the object on sale that are taken from a uniform distribution in the interval $[100, 400]$ (in our run, the first bidder agent has a value of 100 coins, the second of 200, the third of 300 and the fourth of 400);
- each bidder makes offer as fast as it can.

These parameters are equivalent to the hypothesis under which the RET theorem is valid.
In the next subsections, we will show the text output and the Sniffer's graphical output obtained running each auction mechanism implemented and we will examine the results.
In each auction, the name of the auctioneer agent is of the form '`auct_mech`' where $mech$ is the type of auction mechanism while the names of the bidder agents are of the form '`gX_mech`' where X is a integer in the interval $[1, 4]$ and $mech$ is the type of auction mechanism. The complete address will be `name@Vento:1099/JADE` since all the agents are deployed on a single computer called '`Vento`'. In the text output, each agent's output can be recognized by its address at the beginning of the line.

6.3.1 First-price sealed-bid auction

In a first-price sealed-bid auction, the winner of the object pays as much as it offered for it: this means that each bidder must choose between making an high offer (thus incrementing the probability of winning) and getting an high payoff (thus diminishing the probability of winning). Moreover, each bidder can make only one offer: this means that each bidder must try to estimate the private values of the others to beat them.
Supposing that the bidders know that the private values are uniformly distributed, the optimal strategy for each bidder is to offer a bid equal to $(1 - \frac{1}{n})x$ (see Chapter 3), where x represents the private value of the bidder and n is the number of the bidder participating to the auction. This strategy is possible only if the number of participants is common knowledge.
With this strategy, if the bidder's private value is the highest, its bid will be equal to the second-highest private value (given that private values are uniformly distributed among the participants), so it will certainly gain the object and obtain the highest payoff possible.
We programmed all the bidders to offer using this strategy. In fact, as can be seen in Figure 6.1, bidder `g1` offered 75, bidder `g2` offered 150, bidder `g3` offered 225 while `g4` offered 300: clearly, the winner has to be `g4`.
The winner of the auction is agent `g4_fp` with a bid of 300.

In Figure 6.2 it is shown the output of the Sniffer agent. Comparing it with the communication protocol of Figure 5.1 confirms that the requirements of the analysis are fulfilled by the implementation.

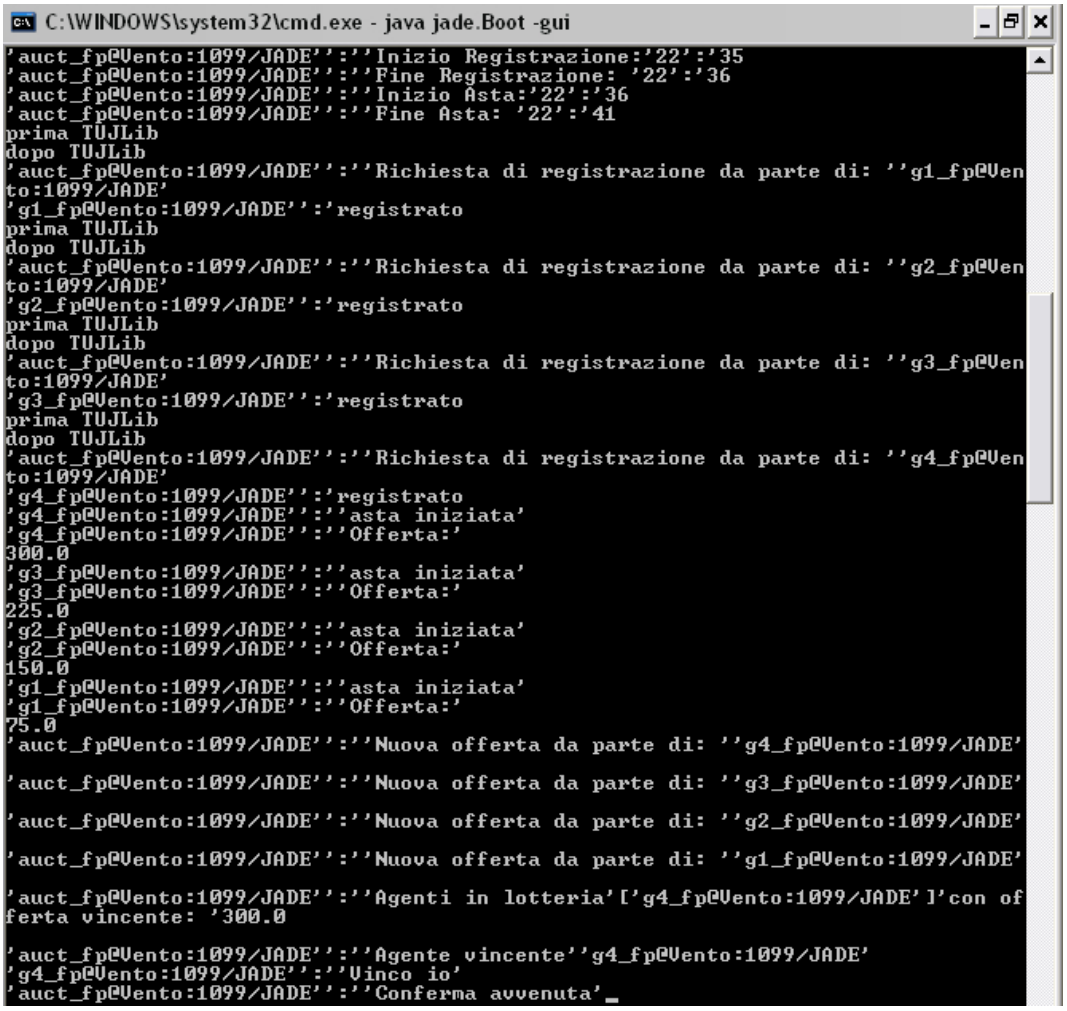

Figure 6.1: First-price sealed-bid shell output.

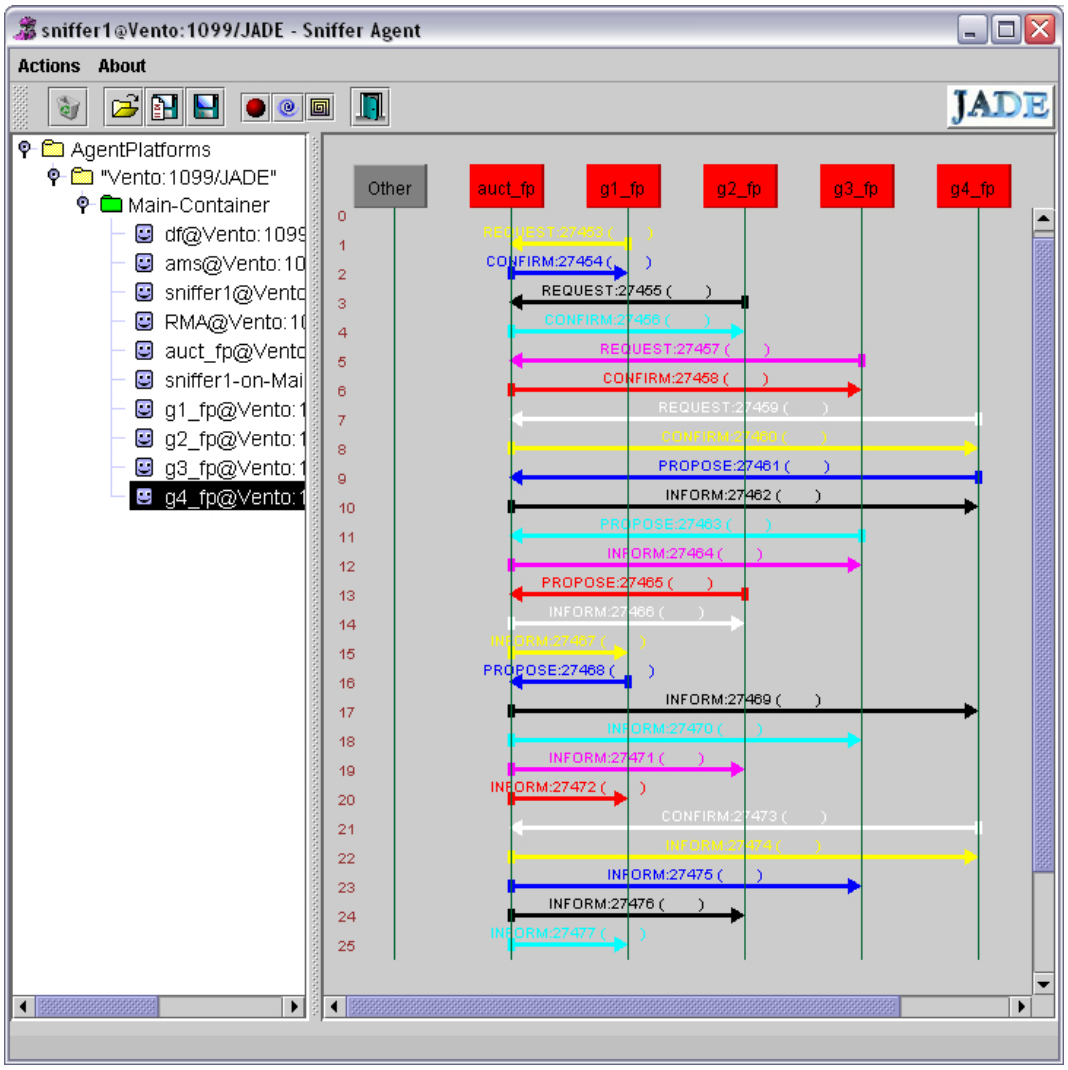

Figure 6.2: Fisrt-price sealed-bid auction: output of Sniffer agent.

6.3.2 Second-price sealed-bid auction

The second-price sealed-bid auction is identical to first-price one, except that the winner pays as much as the highest bid made by the other bidders: this means that the winning bid does not influence the selling price. This pushes the bidders to offer a bids equivalent to their private values (see Chapter 3 for demonstration).
Thus, we used bidders with a truth-telling strategy and the results are shown in Figure 6.3.

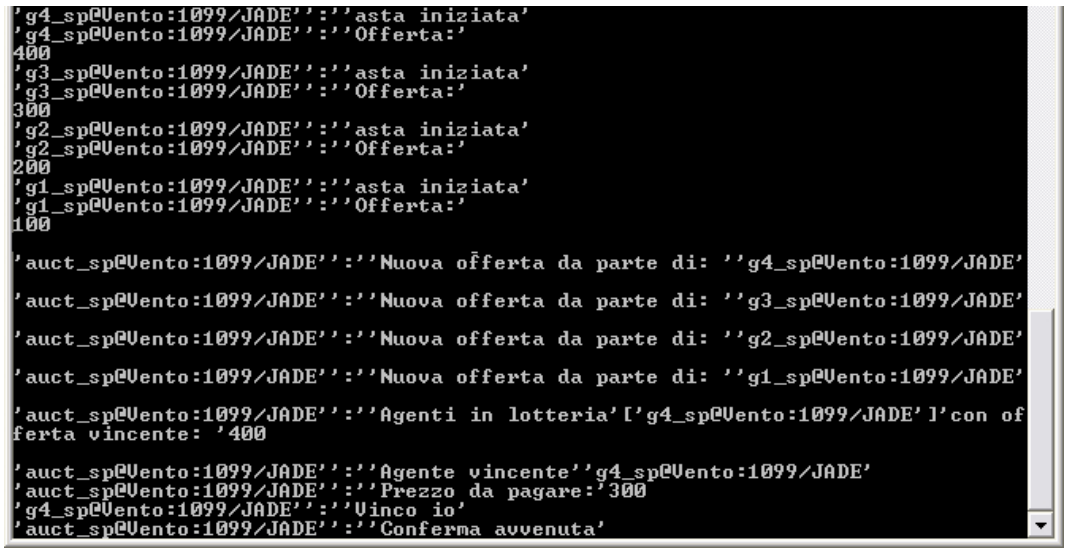

Figure 6.3: Secod-price sealed-bid auction

The winner of the auction is agent `g4` with a bid of 400, while the payment owed is 300.
The output of the sniffer agent is identical to the one of first-price auction (Figure 6.2), thus confirming that, as we said in Chapter 5, the two sealed-bid auction share the same communication protocol.

6.3.3 English auction with continuous bidding

Game theory suggests that, in an English auction with private values, the best strategy for any bidder is to remain in the competition, making small raising, until the price reaches his evaluation of the object, then drop out of the auction: in this way the winner will get the object at a price just a little higher than the second-highest private value (see Section 3.3)
In our implementation, each bidder uses this strategy:

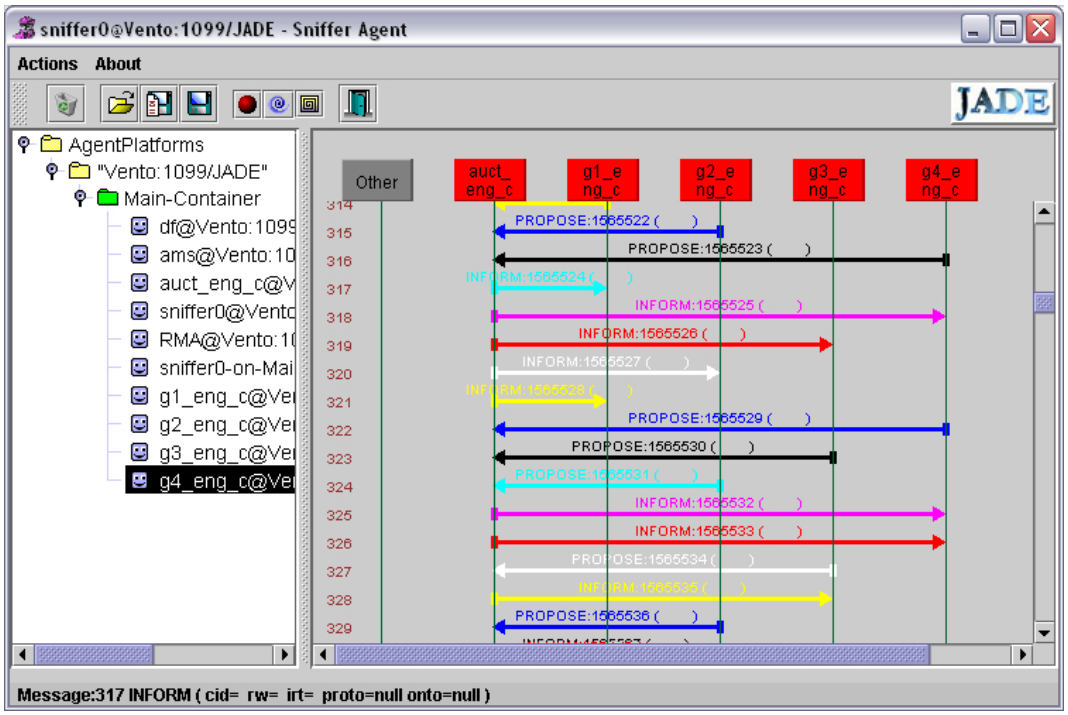

Figure 6.4: English auction with continuous bidding: offering phase

```
eval_offer(New_bid) :-
 object_value(Value), present_bid(Present_bid),
 Present_bid < Value, New_bid is Present_bid + 1,!.
```

As we showed in Chapter 5, the bidder makes the evaluation of a new offer each time the auctioneer inform all the participants that the present winning price is changed. With this strategy, every bidder (except the one who made the last winning bid) makes the same offer as soon as they get the message: being in a English auction with continuous bidding, the auctioneer will accept the first arrived offer as the temporary winning bid (in fact, it bested the old one by 1 coin) and discard all the subsequent identical offer made by other bidders.
The messages on line 318,319,320,321 of Figure 6.4 are `inform` messages that contain the present temporary best offer. As soon as they get this message, all the bidders (except `g1` who was the present winner) send `propose` messages (line 322,323,324) containing new identical offers calculated with the `eval_offer` predicate seen before. The auctioneer gets the first offer (line 322), sees that it is better than the last winning bid and take it as the new winning bid (at line 325): when the auctioneer examines the other offers, it finds that they are *equal* to the

present winning bid, so it discards them.
In Figure 6.5, we can see that, at the end of the auction, agent `g4_eng_c` wins with an offer of 301. We can also note that the auction time was expired before the real conclusion of the competition and this has triggered the extension mechanism that permits to establish the final price of the object; in fact, at the end of the auction time the winner was agent `g3_eng_c` with a bid of 300.
From the theorical point of view, if an English auction has no limits of time, the best selling price will emerge for sure, but a more realistic approach suggests to limit the duration of the auction, like we did. This can create consequences: for example, if the private value of at least two bidder is much bigger than the reservation price, the extended time could expire before the competition is over, thus denying the individuation of the best offer and not attributing the object to the bidder with the highest private value.
This fact is inevitable but we realized that, in this implementation, the order in which the bidders register to the auction influences the order in which they bid, thus giving advantage to a bidder that registered earlier than another: this leads to an unfair attribution of the object.
Hence, we decided to implement another version of the English auction that could change this behavior.

6.3.4 English auction with rounds

The implementation with rounds of an English auction differs from the one with continuous bidding in the way in which equivalent offers are treated: if the auctioneer receives more than one equivalent offer from different bidders in the same round, it takes all of them in consideration, disregarding the order of arrival.
In Figure 6.6, each text line represents a round. We can see that until the offer is less or equal to 100, all four bidders are listed as temporary auction winners, while once the offer becomes greater than 100, the `g1_eng_r` is not listed any more: it has reached its private value. Figure 6.7 confirms this observation: on lines 597,598,599,600 there are still four offers, while in the next round only `g2`,`g3` and `g4` (lines 610,611,612).
This means that, if the auction time expires before the auction arrives to the select a unique winner, all the bidders that have made the last (temporary winning) offer will participate to a lottery to determine the final winner.
In this auction, all the bidders use this offering strategy:

```
eval_offer(New_bid) :-
 num_winners(N), N =\= 1,
```

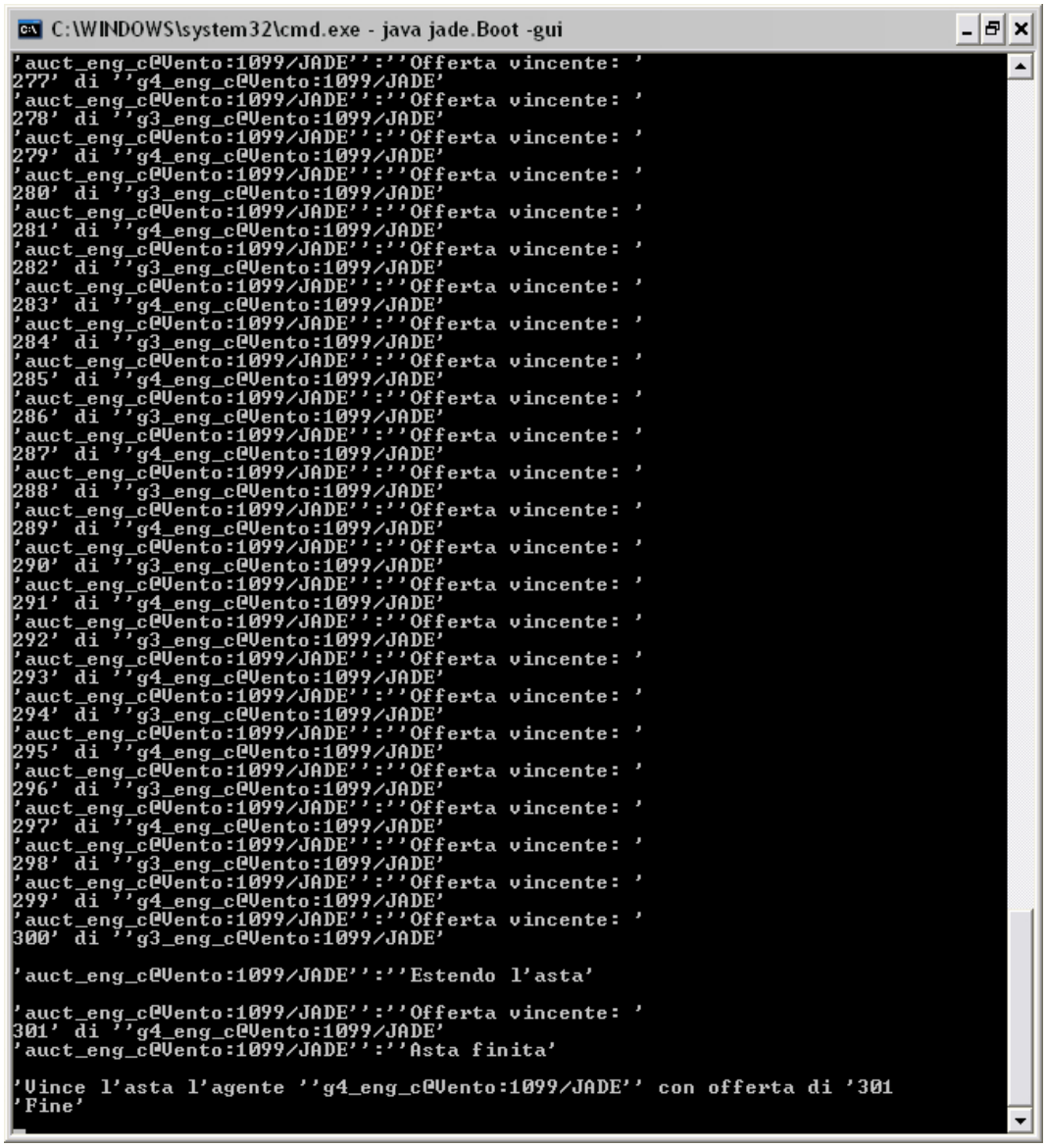

Figure 6.5: English auction with continuous bidding: shell output.

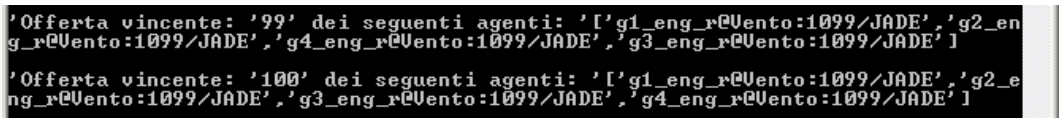

Figure 6.6: English auction with rounds: text output.

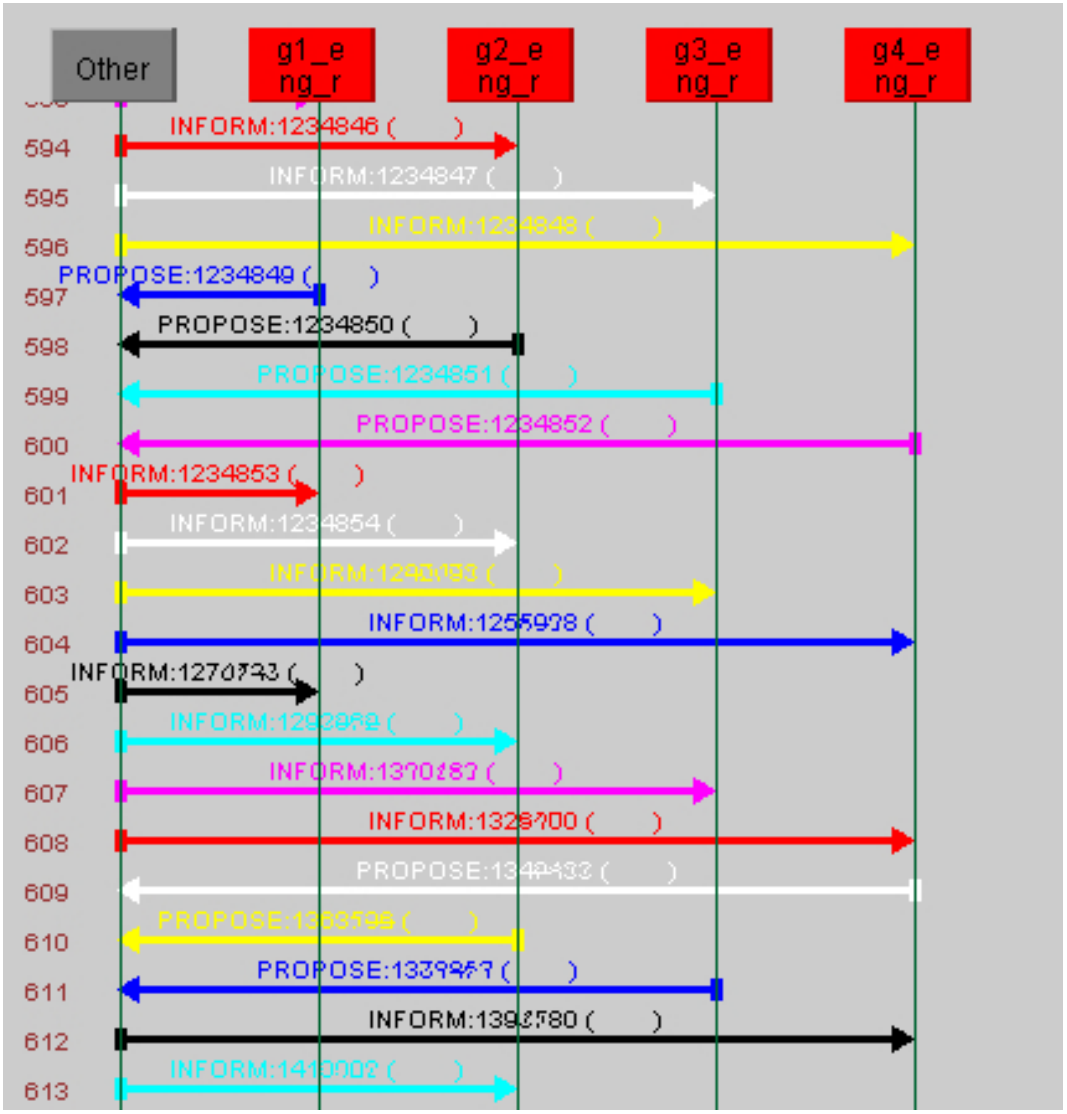

Figure 6.7: English auction with rounds: Sniffer's output

```
 present_bid(Present_bid), object_value(Value),
 Present_bid < Value, New_bid is Present_bid + 1,!.

eval_offer(New_bid) :-
 New_bid = no.
```

If the number of temporary winners (contained in the variable `N`) is different from 1 and the temporary winning bid (contained in the variable `Present_bid`) is strictly less than the private value of the bidder (contained in the variable `Value`), then the new offer of the bidder will be temporary winning bid plus 1, otherwise the bidder will not make any offer (`New_bid = no`).
Note that a bidder using this strategy prefer to make an offer (thus reducing its payoff) to accept a lottery extraction.
In Figure 6.8, we can see that the auction finishes with agent`g4_eng_r` as winner at the best bid of 301.

6.3.5 Results

We run all four auction mechanisms implemented under common conditions to verify the Revenue Equivalence Theorem. Examining all the auction run, we can notice that every one of them terminated with agent `g4_eng_r` as winner, thus demonstrating to be efficient auctions. The two sealed bid mechanisms individuated an auctioneer's revenue of 300, while for the two English mechanisms the revenue was of 301: this difference is caused by the *discrete bidding* strategy that our bidders use. In fact, if the strategy in the English auctions had been to raise the last winning price by 0.1, then the difference between the revenues would have been not 1 but 0.1; if the strategy had been to raise the price by 0.01, then difference would have been 0.01; and so on. Thus, we can say that our implementation verifies the RET.

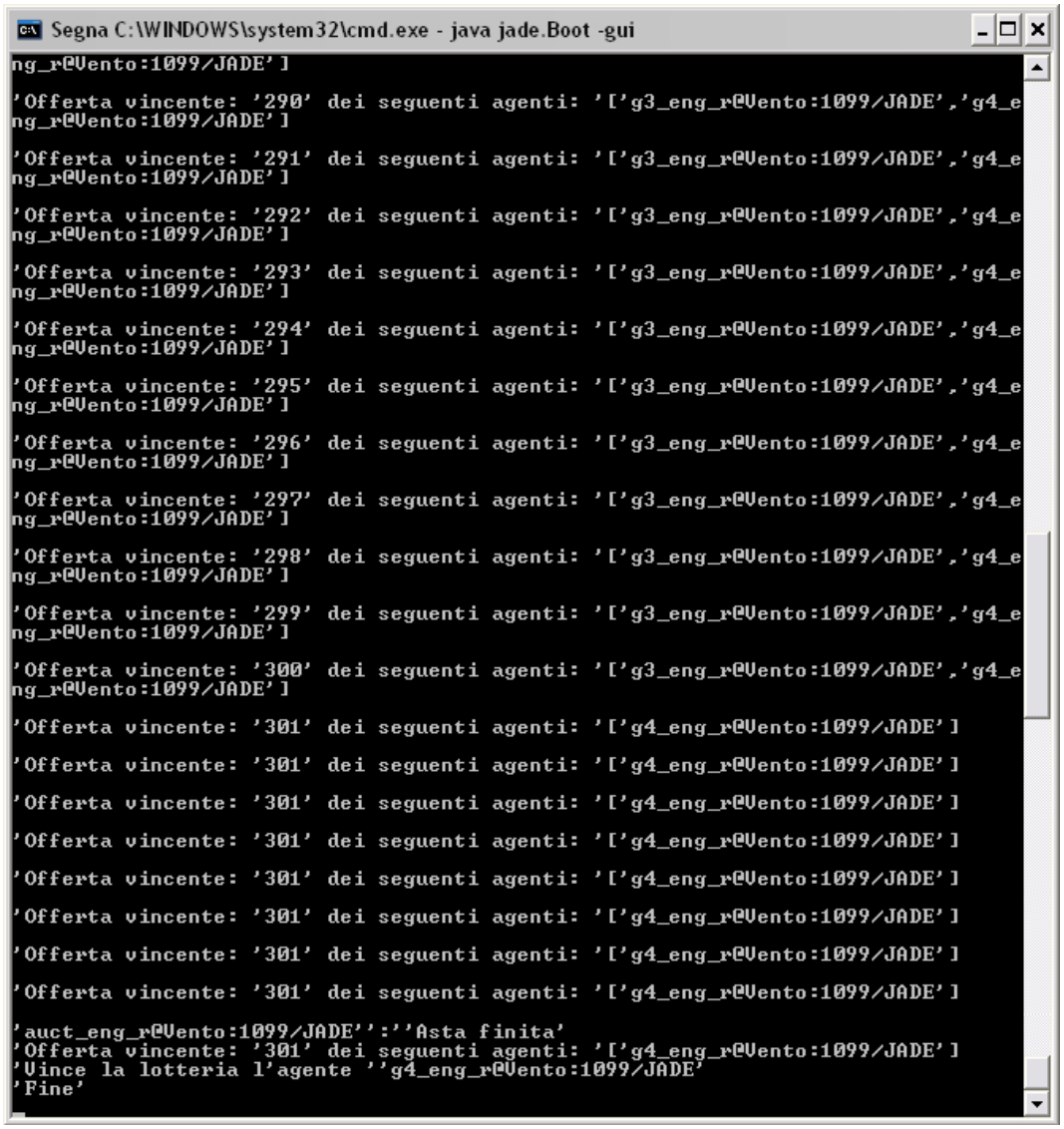

Figure 6.8: Conclusion of English auction with rounds: text output.

Conclusions and future work

In this thesis, we have described the work done to develop a library of agents for simulating auction mechanisms. We have analyzed and implemented four different mechanisms:

- the first-price sealed-bid auction mechanism,
- the second-price sealed-bid auction mechanism,
- the open English auction mechanism with continuous bidding,
- the open English auction mechanism with rounds.

For each auction mechanism, the interaction between auctioneer and bidder has been analyzed and an Interaction Protocol has been produced. In the design phase, the internal behavior of each type of agent has been studied and their customizable features have been highlighted. Each agent's behavior has been written down in a pseudo-Pascal listing.
At last, each agent has been implemented with tuProlog in the DCaseLP environment, thus achieving the goal of providing customizable tools for simulating auction mechanisms. For example, modifying the reservation price of the English auctioneer and the value model of the related bidders, it is possible to simulate English multi-dimensional auctions. Moreover, DCaseLP and JADE supply many tools for analyzing message exchange and debugging agent behaviors, thus helping the user in the analysis of the bidders' strategy.
In the last chapter, we have ran all the implemented mechanism using risk-neutral bidders with independent private value taken from a uniform distribution. Under these hypothesis, Game Theory demonstrated that there exist an optimal bidder's strategy for each of the implemented mechanism: we programmed our test bidders with these strategies and we verified that all the simulated auctions gave the same revenue to the auctioneer and the same payoff to the bidders.
Hence, we can say that the implemented auction mechanisms respect the Revenue Equivalence Theorem.

There are many possibility and directions that can be taken to extend this work. Some of them are briefly described here in the following:

- analyze and implement other less common but interesting auction mechanisms, like double auctions and all-pay auctions,
- build a society of agents, with 'advertising' agents that contain information (like starting and ending time, type of object on sale, type of auction mechanism) on the auctions that are going to be held and 'searcher' agents that look for interesting auction using user's preferences and informs the bidder.
- implement a reputation system, where reliable 'notarial' agents calculates the reputation of the subscribers using other agents' opinions and past behaviors and makes it public to the agent community.

Appendix A

Implemented agents for auction mechanisms

A.1 First-Price Sealed-Bid Auction Agents

Listing A.1: First-Price Sealed-Bid Auctioneer

```
main :-
 startup, registration, get_offer, final.

first_time.
bid_list([]).
present_bid(50).
winner_list([]).
auction_span(5).
reg_span(1).
wait_span(1).
my_name("auct_fp@Vento:1099/JADE").
bidders([]).

agent_msg(Text) :-
 my_name(Name), write(Name), write(":"), write(Text).

broadcast(Performative, TermMsg) :-
 bidders(List),
 pack(TermMsg, Msg),
 multicast(Performative, Msg, List).


multicast(Performative, Message, []).

multicast(Performative, Message, [H|Tail]) :-
```

```
 send(Performative,Message,H),
 multicast(Performative,Message,Tail).


timer(Name,Length,Hour,Min) :-
 assert(start(Name,Hour,Min)),
 Tmp_Finish_time is Min + Length,
 Hour_After is Tmp_Finish_time div 60,
 Finish_Min is Tmp_Finish_time mod 60,
 Finish_H is Hour + Hour_After,
 assert(end(Name,Finish_H,Finish_Min)).


startup :-
 first_time,
 retract(first_time),
 java_object("java.util.Date", [],Date),
 Date<-getMinutes returns Time_Min,
 Date<-getHours returns Time_H,

 reg_span(Reg_Len),
 agent_msg('Inizio Registrazione:'),
 write(Time_H),write(':'),write(Time_Min),nl,
 timer(registration,Reg_Len,Time_H,Time_Min),
 end(registration,End_Reg_H,End_Reg_Min),
 agent_msg('Fine Registrazione: '),
 write(End_Reg_H), write(':'),write(End_Reg_Min),nl,

 auction_span(Auct_Len),
 agent_msg('Inizio Asta:'),
 write(End_Reg_H), write(':'),write(End_Reg_Min),nl,
 timer(auction,Auct_Len,End_Reg_H,End_Reg_Min),
 end(auction,Finish_time_H,Finish_time_Min),
 agent_msg('Fine Asta: '),
 write(Finish_time_H), write(':'),write(Finish_time_Min),nl.

startup.

registration :- end_reg.

registration :-
 java_object("java.util.Date", [],Date),
 Date<-getMinutes returns Now_Min, Date<-getHours returns Now_H,
 end(registration,Fin_H,Fin_Min),check_time(Fin_H,Fin_Min,Now_H,Now_Min),
 handle_msgs,loop2(0,10000),!.

registration :-
```

```
73  assert(end_reg), bidders(BList), length(BList,BList_len),
    end(auction,Finish_time_H,Finish_time_Min),
75  broadcast('INFORM',start(Finish_time_H,Finish_time_Min,BList_len)),
    agent_msg('Fine registrazione'),nl,!.
77
    get_offer :- end_reg, get_offering,!.
79
    get_offer.
81
    get_offering :-
83  java_object("java.util.Date",[],Date),
    end(auction,Finish_time_H,Finish_time_Min),
85  Date<-getMinutes returns Now_min, Date<-getHours returns Now_hour,
    check_time(Finish_time_H,Finish_time_Min,Now_hour,Now_min),
87  handle_msgs,loop2(0,10000),!.

89  get_offering :- finished,!.

91  get_offering :-
    bid_list(BidList),present_bid(Pr_Bid),winner_list(List),
93  max_offer(BidList,Pr_Bid,List),
    present_bid(WinBid),winner_list(WinList),
95  agent_msg("Agenti in lotteria"),
    write(WinList),write("con offerta vincente: "),
97  write(WinBid),nl,nl,
    broadcast('INFORM',winners(WinBid,WinList)),
99  lottery(WinList,Winner),
    assert(winner_is(Winner)),
101 agent_msg("Agente vincente"),write(Winner),nl,
    broadcast('INFORM',real_winner(WinBid,Winner)),
103 assert(finished), assert(wait_confirm), set_wait.

105 set_wait:-
    java_object("java.util.Date",[],Date),
107 Date<-getMinutes returns Time_Min,
    Date<-getHours returns Time_H,
109 wait_span(Wait_Len),
    timer(wait,Wait_Len,Time_H,Time_Min),
111 end(wait,Finish_time_H,Finish_time_Min).

113 final :- finished, wait_confirm, confirm_cycle.

115 final.

117 confirm_cycle :-
    java_object("java.util.Date",[],Date),
119 end(wait,Finish_time_H,Finish_time_Min),
```

```
 Date<-getMinutes returns Now_min, Date<-getHours returns Now_hour,
121 check_time(Finish_time_H,Finish_time_Min,Now_hour,Now_min),
 handle_msgs,loop2(0,10000),!.
123
confirm_cycle :- agent_msg('Nessuna conferma'), retract(wait_confirm).
125
handle_msgs_f :-
127 blocking_receive(Performative, Message, Winner),
 select_f(Performative, Message, Winner),!.
129
handle_msgs_f :- agent_msg("nessun messaggio arrivato").
131
select_f(Performative, Message, Sender) :-
133 bound(Performative),
 bound(Message),
135 unpack(Message,TermMsg),
 handle_f(Performative, TermMsg, Sender).
137
select_f(_,_,_).
139
handle_f('CONFIRM', winner, Sender) :-
141 winner_is(Sender),
 agent_msg("Conferma avvenuta"),!.
143
lottery([],"Nessuno").
145
lottery([Winner],Winner):-!.
147
lottery([X|List],Selected) :-
149 length([X|List],ListLength),
 Seed is ListLength - 1, rand_int(Seed,Num_ext),
151 Position is Num_ext + 1, element(Position,[X|List],Selected).


153
max_offer([(Bid,Bidder)|BList],MaxBid,MaxList) :-
155 Bid > MaxBid, max_offer(BList,Bid,[Bidder]),!.

157 max_offer([(Bid,Bidder)|BList],MaxBid,MaxList) :-
 Bid = MaxBid, append([Bidder],MaxList,NewMaxList),
159 max_offer(BList,MaxBid,NewMaxList),!.

161 max_offer([(Bid,Bidder)|BList],MaxBid,MaxList) :-
 Bid < MaxBid, max_offer(BList,MaxBid,MaxList),!.
163
max_offer([],MaxBid,MaxList) :-
165 retract(present_bid(_)),assert(present_bid(MaxBid)),
 retract(winner_list(_)),assert(winner_list(MaxList)).
```

```
167
loop2(Start,Stop) :- Start =< Stop,!,New_start is Start + 1,
169  loop2(New_start,Stop).

171 loop2(Start,Stop).

173 loop(Num) :-
  retract(conta(Attuale)), Attuale =< Num,!, Nuovo is Attuale + 1,
175  assert(conta(Nuovo)),loop(Num).

177 loop(Num) :- retract(conta(_)).

179
check_time(Fin_h,Fin_min,Now_h,Now_min):- Now_h < Fin_h,!.
181
check_time(Fin_h,Fin_min,Fin_h,Now_min):- Now_min =< Fin_min,!.
183
check_time(X,Y,Z,K) :- fail.
185

187 handle_msgs :-
  receive(Performative, Message, Sender),
189  select(Performative, Message, Sender),!.

191 handle_msgs.

193
select(Performative, Message, Sender) :-
195  bound(Performative),
  bound(Message),
197  unpack(Message,TermMsg),
  handle(Performative, TermMsg, Sender).
199
select(_,_,_) :- true.
201

203 handle('PROPOSE',offer(New_bid),Sender) :-
  agent_msg("Nuova offerta da parte di: "),write(Sender),nl,bid_list(BList),
205  append([(New_bid,Sender)],BList,New_BList),
  retract(bid_list(_)),assert(bid_list(New_BList)),!.
207
handle('REQUEST',register,Sender):-
209  agent_msg("Richiesta di registrazione da parte di: "),
  write(Sender),nl,bidders(List),
211  append([Sender],List,New_List),
  retract(bidders(_)),assert(bidders(New_List)),
213  pack(registered,Msg),send('CONFIRM',Msg,Sender),!.
```

```
handle('CONFIRM', winner, Sender) :-
 wait_confirm,
 winner_is(Sender),
 agent_msg("Conferma avvenuta"), retract(wait_confirm),!.
```

Listing A.2: First-Price Sealed-Bid Bidder

```
main :-
 register,handle_msgs,offering,final.


my_name("g1_fp@Vento:1099/JADE").
object_value(100).
no_reg.

agent_msg(Text) :-
 my_name(Name),write(Name),write(":"),write(Text),nl.

register :- no_reg,
 pack(register,X), send('REQUEST',X,"auct_fp@Vento:1099/JADE"),retract(no_reg).

register.

handle_msgs :-
 blocking_receive(Performative, Message, Sender),
 select(Performative, Message, Sender),!.

handle_msgs :- agent_msg("nessun messaggio").

select(Performative, Message, Sender) :-
 bound(Performative),
 bound(Message),
 unpack(Message,TermMsg),
 handle(Performative, TermMsg, Sender).

select(_,_,_).

handle('CONFIRM',registered,"auct_fp@Vento:1099/JADE") :-
 agent_msg("registrato"),assert(registered),!.

handle('INFORM',start(Time_H,Time_Min,Num_Bidders),"auct_fp@Vento:1099/JADE") :-
 agent_msg("asta iniziata"),
 assert(start(Time_H,Time_Min)), assert(num_bidders(Num_Bidders)),
 assert(can_offer),!.
```

```
38
handle('INFORM',winners(WinBid,WinList),"auct_fp@Vento:1099/JADE") :-
40  assert(winners(WinBid,WinList)),!.

42 handle('INFORM',real_winner(WinBid,Winner),"auct_fp@Vento:1099/JADE") :-
   assert(real_winner(WinBid,Winner)),my_name(Winner),assert(i_win),
44  agent_msg("Vinco io"),retract(start(_,_)).

46 handle(_,_,_).

48 offering :-
   start(_,_), can_offer, registered,
50  eval_offer(New_bid),offer(New_bid),
   agent_msg("Offerta:"),write(New_bid),nl,retract(can_offer),!.
52
offering.
54
eval_offer(New_bid) :-
56  num_bidders(Num_Bidders),object_value(Value),
   New_bid is (Value*(Num_Bidders-1))/Num_Bidders,!.
58


60 offer(Bid) :-
   pack(offer(Bid),X),
62  send('PROPOSE',X,"auct_fp@Vento:1099/JADE").

64 final :- i_win, pack(winner,Msg),
   send('CONFIRM',Msg,"auct_fp@Vento:1099/JADE").
66
final.
```

A.2 Second-Price Sealed-Bid Auction Agents

Listing A.3: Second-Price Sealed-Bid Auctioneer

```
1 main :-
   startup,registration,get_offer,final.
3
first_time.
5 bid_list([]).
present_bid(50).
7 winner_list([]).
```

```
auction_span(1).
reg_span(1).
wait_span(1).
my_name("auct_sp@Vento:1099/JADE").
bidders([]).

agent_msg(Text) :-
 my_name(Name), write(Name), write(":"), write(Text).

broadcast(Performative,TermMsg) :-
 bidders(List),
 pack(TermMsg,Msg),
 multicast(Performative,Msg,List).


multicast(Performative,Message,[]).

multicast(Performative,Message,[H|Tail]) :-
 send(Performative,Message,H),
 multicast(Performative,Message,Tail).


timer(Name,Length,Hour,Min) :-
 assert(start(Name,Hour,Min)),
 Tmp_Finish_time is Min + Length,
 Hour_After is Tmp_Finish_time div 60,
 Finish_Min is Tmp_Finish_time mod 60,
 Finish_H is Hour + Hour_After,
 assert(end(Name,Finish_H,Finish_Min)).


startup :-
 first_time,
 assert(second_bid(0)),
 retract(first_time),
 java_object("java.util.Date", [],Date),
 Date<-getMinutes returns Time_Min,
 Date<-getHours returns Time_H,

 reg_span(Reg_Len),
 agent_msg('Inizio Registrazione:'),
 write(Time_H),write(':'),write(Time_Min),nl,
 timer(registration,Reg_Len,Time_H,Time_Min),
 end(registration,End_Reg_H,End_Reg_Min),
 agent_msg('Fine Registrazione: '),
 write(End_Reg_H), write(':'),write(End_Reg_Min),nl,
```

```
55  auction_span(Auct_Len),
    agent_msg('Inizio Asta:'),
57  write(End_Reg_H), write(':'),write(End_Reg_Min),nl,
    timer(auction,Auct_Len,End_Reg_H,End_Reg_Min),
59  end(auction,Finish_time_H,Finish_time_Min),
    agent_msg('Fine Asta: '),
61  write(Finish_time_H), write(':'),write(Finish_time_Min),nl.

63 startup.

65 registration :- end_reg.

67 registration :-
    java_object("java.util.Date", [],Date),
69  Date<-getMinutes returns Now_Min, Date<-getHours returns Now_H,
    end(registration,Fin_H,Fin_Min),check_time(Fin_H,Fin_Min,Now_H,Now_Min),
71  handle_msgs,loop(0,10000),!.

73 registration :-
    assert(end_reg), bidders(BList), length(BList,BList_len),
75  end(auction,Finish_time_H,Finish_time_Min),
    broadcast('INFORM',start(Finish_time_H,Finish_time_Min,BList_len)),
77  agent_msg('Fine registrazione'),nl,!.

79 get_offer :- end_reg, get_offering,!.

81 get_offer.

83 get_offering :-
    java_object("java.util.Date", [],Date),
85  end(auction,Finish_time_H,Finish_time_Min),
    Date<-getMinutes returns Now_min, Date<-getHours returns Now_hour,
87  check_time(Finish_time_H,Finish_time_Min,Now_hour,Now_min),
    handle_msgs,loop(0,10000),!.
89
   get_offering :- finished,!.
91
   get_offering :-
93  bid_list(BidList),present_bid(Pr_Bid),winner_list(List),
    max_offer(BidList,Pr_Bid,0,List),
95  present_bid(WinBid),winner_list(WinList),
    agent_msg("Agenti in lotteria"),
97  write(WinList),write("con offerta vincente: "),
    write(WinBid),nl,nl,
99  broadcast('INFORM',winners(WinBid,WinList)),
    lottery(WinList,Winner),
101 assert(winner_is(Winner)),
```

```
 agent_msg("Agente vincente"),write(Winner),nl,
103 second_bid(SecBid),agent_msg("Prezzo da pagare:"),write(SecBid),nl,
 broadcast('INFORM',real_winner(WinBid,SecBid,Winner)),
105 assert(finished), assert(wait_confirm), set_wait.

107 set_wait:-
 java_object("java.util.Date", [],Date),
109 Date<-getMinutes returns Time_Min,
 Date<-getHours returns Time_H,
111 wait_span(Wait_Len),
 timer(wait,Wait_Len,Time_H,Time_Min),
113 end(wait,Finish_time_H,Finish_time_Min).

115 final :- finished, wait_confirm, confirm_cycle.

117 final.

119 confirm_cycle :-
 java_object("java.util.Date", [],Date),
121 end(wait,Finish_time_H,Finish_time_Min),
 Date<-getMinutes returns Now_min, Date<-getHours returns Now_hour,
123 check_time(Finish_time_H,Finish_time_Min,Now_hour,Now_min),
 handle_msgs,loop(0,10000),!.
125
 confirm_cycle :- agent_msg('Nessuna conferma'), retract(wait_confirm).
127
 handle_msgs_f :-
129 blocking_receive(Performative, Message, Winner),
 select_f(Performative, Message, Winner),!.
131
 handle_msgs_f :- agent_msg("nessun messaggio arrivato").
133
 select_f(Performative, Message, Sender) :-
135 bound(Performative),
 bound(Message),
137 unpack(Message,TermMsg),
 handle_f(Performative, TermMsg, Sender).
139
 select_f(_,_,_).
141
 handle_f('CONFIRM', winner, Sender) :-
143 winner_is(Sender),
 agent_msg("Conferma avvenuta"),!.
145
 lottery([],"Nessuno") :-
147 agent_msg("Non e' stato raggiunto il prezzo di riserva"),nl.
```

```
149 lottery([Winner],Winner):-!.

151 lottery([X|List],Selected) :-
     length([X|List],ListLength),
153  Seed is ListLength - 1, rand_int(Seed,Num_ext),
     Position is Num_ext + 1, element(Position,[X|List],Selected).
155

157 max_offer([(Bid,Bidder)|BList],MaxBid,SecBid,MaxList) :-
     Bid > MaxBid, max_offer(BList,Bid,MaxBid,[Bidder]),!.
159
    max_offer([(Bid,Bidder)|BList],MaxBid,SecBid,MaxList) :-
161  Bid = MaxBid, append([Bidder],MaxList,NewMaxList),
     max_offer(BList,MaxBid,MaxBid,NewMaxList),!.
163
    max_offer([(Bid,Bidder)|BList],MaxBid,SecBid,MaxList) :-
165  Bid < MaxBid, Bid =< SecBid, max_offer(BList,MaxBid,SecBid,MaxList),!.

167 max_offer([(Bid,Bidder)|BList],MaxBid,SecBid,MaxList) :-
     Bid < MaxBid, Bid > SecBid, max_offer(BList,MaxBid,Bid,MaxList),!.
169
    max_offer([],MaxBid,SecBid,MaxList) :-
171  retract(second_bid(_)),assert(second_bid(SecBid)),
     retract(present_bid(_)),assert(present_bid(MaxBid)),
173  retract(winner_list(_)),assert(winner_list(MaxList)).

175 loop(Start,Stop) :- Start =< Stop,!,New_start is Start + 1,
     loop(New_start,Stop).
177
    loop(Start,Stop).
179
    check_time(Fin_h,Fin_min,Now_h,Now_min):- Now_h < Fin_h,!.
181
    check_time(Fin_h,Fin_min,Fin_h,Now_min):- Now_min =< Fin_min,!.
183
    check_time(X,Y,Z,K) :- fail.
185

187 handle_msgs :-
     receive(Performative, Message, Sender),
189  select(Performative, Message, Sender),!.

191 handle_msgs.


193
    select(Performative, Message, Sender) :-
195  bound(Performative),
```

```
 bound(Message),
 unpack(Message,TermMsg),
 handle(Performative, TermMsg, Sender).

select(_,_,_) :- true.


handle('PROPOSE',offer(New_bid),Sender) :-
 agent_msg("Nuova offerta da parte di: "),write(Sender),nl,bid_list(BList),
 append([(New_bid,Sender)],BList,New_BList),
 retract(bid_list(_)),assert(bid_list(New_BList)),!.

handle('REQUEST',register,Sender):-
 agent_msg("Richiesta di registrazione da parte di: "),
 write(Sender),nl,bidders(List),
 append([Sender],List,New_List),
 retract(bidders(_)),assert(bidders(New_List)),
 pack(registered,Msg),send('CONFIRM',Msg,Sender),!.

handle('CONFIRM', winner, Sender) :-
 wait_confirm,
 winner_is(Sender),
 agent_msg("Conferma avvenuta"), retract(wait_confirm),!.
```

Listing A.4: Second-Price Sealed-Bid Bidder

```
main :-
 register,handle_msgs,offering,final.


my_name("g1_sp@Vento:1099/JADE").
object_value(100).
no_reg.

agent_msg(Text) :-
 my_name(Name),write(Name),write(":"),write(Text),nl.

register :- no_reg,
 pack(register,X), send('REQUEST',X,"auct_sp@Vento:1099/JADE"),retract(no_reg).

register.

handle_msgs :-
 blocking_receive(Performative, Message, Sender),
 select(Performative, Message, Sender),!.
```

```
20
handle_msgs :- agent_msg("nessun messaggio").
22
select(Performative, Message, Sender) :-
24 bound(Performative),
 bound(Message),
26 unpack(Message,TermMsg),
 handle(Performative, TermMsg, Sender).
28
select(_,_,_).
30
handle('CONFIRM',registered,"auct_sp@Vento:1099/JADE") :-
32 agent_msg("registrato"),assert(registered),!.

34 handle('INFORM',start(Time_H,Time_Min,Num_Bidders),"auct_sp@Vento:1099/JADE") :-
 agent_msg("asta iniziata"),
36 assert(start(Time_H,Time_Min)), assert(num_bidders(Num_Bidders)),
 assert(can_offer),!.
38
handle('INFORM',winners(WinBid,WinList),"auct_sp@Vento:1099/JADE") :-
40 assert(winners(WinBid,WinList)),!.

42 handle('INFORM',real_winner(WinBid,Price,Winner),"auct_sp@Vento:1099/JADE") :-
 assert(real_winner(WinBid,Price,Winner)),my_name(Winner),assert(i_win),
44 agent_msg("Vinco io"),retract(start(_,_)).

46 handle(_,_,_).

48 offering :-
 start(_,_), can_offer, registered,
50 num_bidders(Num_Bidders),eval_offer(New_bid),offer(New_bid),
 agent_msg("Offerta:"),write(New_bid),nl,retract(can_offer),!.
52
offering.
54
eval_offer(New_bid) :-
56 object_value(Value), New_bid is Value,!.


58
offer(Bid) :-
60 pack(offer(Bid),X),
 send('PROPOSE',X,"auct_sp@Vento:1099/JADE").
62
final :- i_win, pack(winner,Msg), send('CONFIRM',Msg,"auct_sp@Vento:1099/JADE").
64
final.
```

A.3 English auction with continuous bidding

Listing A.5: English auction with continuous bidding: Auctioneer

```
2  main :-
     startup, registration, get_offer, final.
4
6  reserve_price(50).
   reg_span(1).
8  auction_span(2).
   alarm_span(1).
10 extension_span(3).

12 first_time.
   my_name("auct_eng_c@Vento:1099/JADE").
14 present_winner(nobody).
   propose(no).
16 n_alarm(0).
   bidders([]).
18
   agent_msg(Text) :-
20   my_name(Name), write(Name), write(":"), write(Text), nl.

22 agent_write(Text):-
     write(Text).
24
   broadcast(Performative,TermMsg) :-
26   bidders(List),
     pack(TermMsg,Msg),
28   multicast(Performative,Msg,List).


30
   multicast(Performative,Message,[]).
32
   multicast(Performative,Message,[H|Tail]) :-
34   send(Performative,Message,H),
     multicast(Performative,Message,Tail).
36
   timer(Name,Length,Hour,Min) :-
38   assert(start(Name,Hour,Min)),
```

```
 Tmp_Finish_time is Min + Length,
 Hour_After is Tmp_Finish_time div 60,
 Finish_Min is Tmp_Finish_time mod 60,
 Finish_H is (Hour + Hour_After) mod 24,
 assert(end(Name,Finish_H,Finish_Min)).

startup :-
 first_time,
 retract(first_time),

 java_object("java.util.Date", [],Date),
 Date<-getMinutes returns Time_Min,
 Date<-getHours returns Time_H,
 agent_msg('Inizio Registrazione:'),
 agent_write(Time_H),agent_write(':'),agent_write(Time_Min),nl,

 reg_span(Reg_Len),
 timer(registration,Reg_Len,Time_H,Time_Min),
 end(registration,End_Reg_H,End_Reg_Min),
 agent_msg('Inizio Asta: '),
 write(End_Reg_H), write(':'),write(End_Reg_Min),nl,

 auction_span(Len),
 timer(auction,Len,End_Reg_H,End_Reg_Min),
 end(auction,Finish_H,Finish_Min),
 agent_msg('Fine asta: '),
 agent_write(Finish_H), agent_write(':'),agent_write(Finish_Min),nl,

 alarm_span(Alarm),
 timer(alarm,Len - Alarm,End_Reg_H,End_Reg_Min),
 end(alarm,Alarm_H,Alarm_Min),
 agent_msg('Allarme: '),
 agent_write(Alarm_H), agent_write(':'),agent_write(Alarm_Min),nl,

 extension_span(Len_ext),
 timer(ext,Len + Len_ext,End_Reg_H,End_Reg_Min),
 end(ext,Ext_time_H,Ext_time_Min),
 agent_msg('Estensione: '),
 agent_write(Ext_time_H), agent_write(':'),agent_write(Ext_time_Min),nl,!.

startup.

registration :- end_reg.

registration :-
 java_object("java.util.Date", [],Date),
 Date<-getMinutes returns Now_Min, Date<-getHours returns Now_H,
```

```
86  end(registration,Fin_H,Fin_Min),check_time(Fin_H,Fin_Min,Now_H,Now_Min),
    handle_msgs,loop2(1,100000),!.
88
    registration :-
90  reserve_price(Pr), broadcast('INFORM',reserve_price(Pr)),
    assert(present_bid(Pr)),
92  bidders(BList), length(BList,BList_len),
    end(auction,Finish_time_H,Finish_time_Min),
94  broadcast('INFORM',start(Finish_time_H,Finish_time_Min,BList_len)),
    assert(end_reg),assert(auct_going),
96  agent_msg('Fine registrazione'),nl,!.

98  get_offer :- auct_going, get_offering,!.

100 get_offer.

102 get_offering :-
    java_object("java.util.Date",[],Date),
104 Date<-getMinutes returns Now_min,
    Date<-getHours returns Now_hour,
106 end(auction,Finish_time_H,Finish_time_Min),
    check_time(Finish_time_H,Finish_time_Min,Now_hour,Now_min),
108 handle_msgs,
    check_alarm(Now_hour,Now_min),
110 retract(propose(_)),
    assert(propose(no)),
112 loop2(1,10000),!.

114 get_offering :-
    n_alarm(X),X =\= 0,retract(n_alarm(_)),nl,
116 agent_msg("Estendo l'asta"),nl,
    retract(end(auction,Finish_time_H,Finish_time_Min)),
118 end(ext,Ext_time_H,Ext_time_Min),
    assert(end(auction,Ext_time_H,Ext_time_Min)),!.
120
    get_offering :-
122 agent_msg("Asta finita"),retract(auct_going),assert(auct_end).

124 get_offering.

126 loop2(Start,Stop) :- Start =< Stop,!,New_start is Start + 1,
    loop2(New_start,Stop).
128
    loop2(Start,Stop).
130


132 check_time(Fin_h,Fin_min,Now_h,Now_min):- Fin_h > Now_h,!.
```

```
134 check_time(Fin_h,Fin_min,Fin_h,Now_min):- Fin_min >= Now_min,!.

136 check_time(0,_,Now_h,_):- Now_h >= 12.

138 check_time(X,Y,Z,K) :- fail.

140

    check_alarm(Time_h,Time_min) :-
142  propose(yes), end(alarm,Alarm_H,Alarm_Min),
     check_time(Time_h,Time_min,Alarm_H,Alarm_Min),
144  retract(n_alarm(Num)), New_num is Num + 1, assert(n_alarm(New_num)),!.

146 check_alarm(_,_).

148

    handle_msgs :-
150  receive(Performative, Message, Sender),
     select(Performative, Message, Sender),
152  !.

154 handle_msgs.

156

    select(Performative, Message, Sender) :-
158  bound(Performative),
     bound(Message),
160  unpack(Message,TermMsg),
     handle(Performative, TermMsg, Sender).
162

    select(_,_,_) :- true.
164



166

    handle('PROPOSE',offer(New_bid),Sender) :-
168  retract(propose(_)),
     assert(propose(yes)),
170  present_bid(Pr_bid),
     compare(New_bid,Pr_bid,Sender).
172

    handle('REQUEST',register,Sender):-
174  agent_msg("Richiesta di registrazione da parte di: "),
     write(Sender),nl,bidders(List),
176  append([Sender],List,New_List),
     retract(bidders(_)),assert(bidders(New_List)),
178  pack(registered,Msg),send('CONFIRM',Msg,Sender),!.
```

```
180
182 compare(New,Old,Sender) :-
     better_bid(New,Old),
184  retract(present_bid(_)),assert(present_bid(New)),
     retract(present_winner(_)),assert(present_winner(Sender)),
186  agent_msg("Offerta vincente: "),agent_write(New),agent_write(" di "),
     agent_write(Sender), nl,
188  pack(you_win,Msg),send('INFORM',Msg,Sender),
     broadcast('INFORM',present_bid(New)),!.
190
    compare(New,Old,Sender).
192
    better_bid(NewB,OldB) :-  NewB > OldB.
194
    final :- auct_end,
196  reserve_price(Res_pri),present_bid(Bid), Res_pri > Bid,
     present_winner(Winner),
198  broadcast('INFORM',auct_end), retract(auct_end),
     agent_msg("La migliore offerta non ha raggiunto il prezzo di riserva."),
200  Bid = 0,
     broadcast('INFORM',real_winner(Winner,Bid)),write("Fine Asta"),nl,!.
202
    final :- auct_end,
204  present_bid(Bid), present_winner(Winner),
     broadcast('INFORM',auct_end), retract(auct_end),
206  nl,agent_write("Vince l'asta l'agente "),
     write(Winner),write(" con offerta di "),write(Bid),nl,
208  broadcast('INFORM',real_winner(Winner,Bid)),write("Fine"),nl.

210 final.
```

Listing A.6: English auction with continuous bidding: Bidder

```
    main :-
2    register,handle_msgs,offering.

4   my_name("g1_eng_c@Vento:1099/JADE").
    i_win(false).
6   object_value(100).
    no_reg.
8
    agent_msg(Text) :-
10   my_name(Name),write(Name),write(":"),write(Text),nl.
```

```
register :- no_reg,
 pack(register,X), send('REQUEST',X,"auct_eng_c@Vento:1099/JADE"),
 retract(no_reg).

register.

handle_msgs :-
 blocking_receive(Performative, Message, Sender),
 select(Performative, Message, Sender),!.

handle_msgs.

select(Performative, Message, Sender) :-
 bound(Performative),
 bound(Message),
 unpack(Message,TermMsg),
 handle(Performative, TermMsg, Sender).

select(_,_,_).

handle('CONFIRM',registered,"auct_eng_c@Vento:1099/JADE") :-
 agent_msg("registrato"),assert(registered),!.

handle('INFORM',reserve_price(Pr),"auct_eng_c@Vento:1099/JADE"):-
 assert(present_bid(Pr)).

handle('INFORM',
start(Finish_time_H,Finish_time_Min,BList_len),
"auct_eng_c@Vento:1099/JADE") :-
 assert(start(Finish_time_H,Finish_time_Min)),
 assert(num_bidders(Num_Bidders)),!.

handle('INFORM',present_bid(Bid),"auct_eng_c@Vento:1099/JADE") :-
 change_bid(Bid),
 retract(i_win(_)),assert(i_win(false)),!.

handle('INFORM',you_win,"auct_eng_c@Vento:1099/JADE") :-
 retract(i_win(_)),assert(i_win(true)), get_w_bid,!.

handle(_,_,_).

change_bid(Bid) :-
   retract(present_bid(_)),assert(present_bid(Bid)).

get_w_bid :-
   blocking_receive(Performative, Message, Sender), bound(Performative),
   bound(Message),unpack(Message,TermMsg),get_w_aux(TermMsg),!.
```

```
get_w_aux(present_bid(Bid)) :- change_bid(Bid).

offering :-
 registered, start(_,_), i_win(false),
 eval_offer(New_bid), offer(New_bid),!.

offering.

eval_offer(New_bid) :-
 object_value(Value), present_bid(Present_bid), Present_bid < Value,
 New_bid is Present_bid + 1,!.

eval_offer(Present_bid,New_bid) :- New_bid = no,
 agent_msg("non posso permettermelo").

offer(no):-!.

offer(Bid) :-
 pack(offer(Bid),X),
 send('PROPOSE',X,"auct_eng_c@Vento:1099/JADE").
```

A.4 English auction with rounds

Listing A.7: English auction with rounds: Auctioneer

```
main :-
 startup,registration,get_offer,final.


reserve_price(50).
reg_span(1).
auction_span(5).
alarm_span(1).
extension_span(2).
time_step(1).

first_time.
my_name("auct_eng_r@Vento:1099/JADE").
propose(no).
n_alarm(0).
```

```
bidders([]).
present_winner([]).

agent_msg(Text) :-
 my_name(Name), write(Name), write(":"), write(Text), nl.

agent_write(Text):-
 write(Text).

broadcast(Performative,TermMsg) :-
 bidders(List),
 multicast(Performative,TermMsg,List).

multicast(Performative,Message,[]).

multicast(Performative,Message,[H|Tail]) :-
 pack(Message,Msg),
 send(Performative,Msg,H),
 multicast(Performative,Message,Tail).

timer(Name,Length,Hour,Min) :-
 assert(start(Name,Hour,Min)),
 Tmp_Finish_time is Min + Length,
 Hour_After is Tmp_Finish_time div 60,
 Finish_Min is Tmp_Finish_time mod 60,
 Finish_H is (Hour + Hour_After) mod 24,
 assert(end(Name,Finish_H,Finish_Min)).


startup :-
 first_time,
 retract(first_time),

 java_object("java.util.Date", [],Date),
 Date<-getMinutes returns Time_Min,
 Date<-getHours returns Time_H,
 agent_msg('Inizio Registrazione:'),
 agent_write(Time_H),agent_write(':'),agent_write(Time_Min),nl,

 reg_span(Reg_Len),
 timer(registration,Reg_Len,Time_H,Time_Min),
 end(registration,End_Reg_H,End_Reg_Min),
 agent_msg('Inizio Asta: '),
 write(End_Reg_H), write(':'),write(End_Reg_Min),nl,

 auction_span(Len),
 timer(auction,Len,End_Reg_H,End_Reg_Min),
```

```
 end(auction,Finish_H,Finish_Min),
65 agent_msg('Fine asta: '),
 agent_write(Finish_H), agent_write(':'),agent_write(Finish_Min),nl,
67
 alarm_span(Alarm),
69 timer(alarm,Len - Alarm,End_Reg_H,End_Reg_Min),
 end(alarm,Alarm_H,Alarm_Min),
71 agent_msg('Allarme: '),
 agent_write(Alarm_H), agent_write(':'),agent_write(Alarm_Min),nl,
73
 extension_span(Len_ext),
75 timer(ext,Len + Len_ext,End_Reg_H,End_Reg_Min),
 end(ext,Ext_time_H,Ext_time_Min),
77 agent_msg('Estensione: '),
 agent_write(Ext_time_H), agent_write(':'),agent_write(Ext_time_Min),nl,!.
79

81 startup.

83 registration :- end_reg.

85 registration :-
 java_object("java.util.Date", [],Date),
87 Date<-getMinutes returns Now_Min, Date<-getHours returns Now_H,
 end(registration,Fin_H,Fin_Min),check_time(Fin_H,Fin_Min,Now_H,Now_Min),
89 handle_msgs,loop2(1,100000),!.

91 registration :-
 reserve_price(Pr), broadcast('INFORM',reserve_price(Pr)),
93 assert(present_bid(Pr)),
 bidders(BList), length(BList,BList_len),
95 end(auction,Finish_time_H,Finish_time_Min),
 broadcast('INFORM',start(Finish_time_H,Finish_time_Min,BList_len)),
97 assert(end_reg),assert(auct_going),
 agent_msg('Fine registrazione '),nl,!.
99

101 get_offer :-
 loop(1,10000),
103 handle_msgs([],OfferList),present_bid(Pr_Bid),present_winner(List),
 max_offer(OfferList,Pr_Bid,List),
105  java_object("java.util.Date", [],Date),
 end(auction,Finish_time_H,Finish_time_Min),
107 Date<-getMinutes returns Now_min,
 Date<-getHours returns Now_hour,
109 check_alarm(Now_hour,Now_min),
 check_time(Finish_time_H,Finish_time_Min,Now_hour,Now_min),
```

```
111  retract(propose(_)),
     assert(propose(no)),
113  !.

115 get_offer :-
     n_alarm(X),X =\= 0,retract(n_alarm(_)),
117  agent_msg("Estendo l'asta"),nl,
     retract(end(auction,Finish_time_H,Finish_time_Min)),
119  end(ext,Ext_time_H,Ext_time_Min),
     assert(end(auction,Ext_time_H,Ext_time_Min)),!.
121
    get_offer :- auct_going,nl,agent_msg("Asta finita"),
123 retract(auct_going),assert(auct_end).

125 get_offer.

127 loop(Start,Stop) :- Start =< Stop,!,New_start is Start + 1,
     loop(New_start,Stop).
129
    loop(Start,Stop).
131

133 waiting(Date,End_hour,End_min) :-
     loop(1000),
135  Date<-getMinutes returns Now_min,Date<-getHours returns Now_hour,
     check_time(End_hour,End_min,Now_hour,Now_min),
137  waiting(Date,End_hour,End_min).

139 waiting(_,_,_).

141
    check_time(Fin_h,Fin_min,Now_h,Now_min):- Fin_h > Now_h,!.
143
    check_time(Fin_h,Fin_min,Fin_h,Now_min):- Fin_min >= Now_min,!.
145
    check_time(X,Y,Z,K) :- fail.
147

149 check_alarm(Time_h,Time_min) :- propose(yes),
     m_alarm(Alarm_min), h_alarm(Alarm_h),
151  check_time(Time_h,Time_min,Alarm_h,Alarm_min),
     retract(n_alarm(Num)),New_num is Num + 1, assert(n_alarm(New_num)),nl,
153  write("Allarme: "),agent_write(New_num),nl,!.

155 check_alarm(_,_).

157
```

```
handle_msgs(List,ListRes) :-
 receive(Performative, Message, Sender),
 select(Performative, Message, Sender,List,ListApp),
 handle_msgs(ListApp,ListRes),!.

handle_msgs(List,List).


select(Performative, Message, Sender, L, LRes) :-
 bound(Performative),
 bound(Message),
 unpack(Message,TermMsg),
 handle(Performative, TermMsg, Sender, L, LRes).

select(_,_,_) :- true.


handle('PROPOSE',offer(New_bid),Sender,L,LRes) :-
 retract(propose(_)),
 assert(propose(yes)),
 append([(New_bid,Sender)],L,LRes).

handle('REQUEST',register,Sender,L,LRes):-
 agent_msg("Richiesta di registrazione da parte di: "),
 write(Sender),nl,bidders(List),
 append([Sender],List,New_List),
 retract(bidders(_)),assert(bidders(New_List)),
 pack(registered,Msg),send('CONFIRM',Msg,Sender),!.

eval_offer([]).

eval_offer([(NewBid,Sender)|List]):-
 present_bid(PrBid), better_bid(NewBid,PrBid),
 retract(present_bid(_)),assert(present_bid(NewBid)),
 retract(present_winner(_)),assert(present_winner([Sender])),
 eval_offer(List),!.

eval_offer([(NewBid,Sender)|List]):-
 present_bid(PrBid), equal_bid(NewBid,PrBid),
 retract(present_winner(WinnerList)),
 append([Sender],WinnerList,NewWinnerList),
 assert(present_winner(NewWinnerList)),
 eval_offer(List),!.

eval_offer([(NewBid,Sender)|List]):- eval_offer(List),!.
```

```
205 better_bid(NewBid,OldBid) :- NewBid > OldBid.

207

    equal_bid(NewBid,OldBid) :- NewBid == OldBid.
209

211 max_offer([(Bid,Bidder)|BList],MaxBid,MaxList) :-
     Bid > MaxBid, max_offer(BList,Bid,[Bidder]),!.
213
    max_offer([(Bid,Bidder)|BList],MaxBid,MaxList) :-
215  Bid = MaxBid, append([Bidder],MaxList,NewMaxList),
     max_offer(BList,MaxBid,NewMaxList),!.
217
    max_offer([(Bid,Bidder)|BList],MaxBid,MaxList) :-
219  Bid < MaxBid, max_offer(BList,MaxBid,MaxList),!.

221 max_offer([],MaxBid,MaxList) :-
     retract(present_bid(_)),assert(present_bid(MaxBid)),
223  retract(present_winner(_)),assert(present_winner(MaxList)).


225
    final :- auct_going,
227  reserve_price(null),present_bid(Bid),
     agent_msg("Offerta vincente: "),agent_write(Bid),nl,
229  present_winner(Winners),
     agent_write(" dei seguenti agenti: "),agent_write(Winners), nl,!.
231
    final :- auct_end,
233  reserve_price(Res_pri),present_bid(Bid), Res_pri > Bid,
      agent_msg("La migliore offerta non ha raggiunto il prezzo di riserva."),
235  broadcast('INFORM',auct_end),retract(auct_end),!.

237 final :- auct_going,
     present_bid(Bid),agent_write("Offerta vincente: "),agent_write(Bid),
239  present_winner(Winners),
     agent_write(" dei seguenti agenti: "),agent_write(Winners),nl,nl,
241  length(Winners,Num_win),multicast('INFORM',you_win(Num_win,Bid),Winners),
     broadcast('INFORM',present_bid(Bid)).
243
    final :- auct_end,
245  present_bid(Bid),agent_write("Offerta vincente: "),agent_write(Bid),
     present_winner(Winners),
247  agent_write(" dei seguenti agenti: "),agent_write(Winners),nl,
     broadcast('INFORM',auct_end),retract(auct_end),
249  length(Winners,Num_win),multicast('INFORM',last_win(Num_win,Bid),Winners),
     agent_write("Vince la lotteria l'agente "),
251  lottery(Winners,Winner),write(Winner),nl,
```

```
 broadcast('INFORM',real_winner(Winner,Bid)),write("Fine"),nl.
253
final.
255


257 lottery([],"Nessuno").

259 lottery([Winner],Winner).

261 lottery([X|List],Selected) :-
 length([X|List],ListLength), Seed is ListLength - 1,
263  rand_int(Seed,Num_ext), Position is Num_ext + 1,
 element(Position,[X|List],Selected).
```

Listing A.8: English auction with rounds: Bidder

```
main :-
2  handle_msgs,offering.


4
i_win(false).
6 my_name("g1_eng_r@Vento:1099/JADE").
object_value(100).
8 num_winners(0).

10 agent_msg(Text) :-
 my_name(Name),write(Name),write(":"),write(Text).
12


14 handle_msgs :-
 blocking_receive(Performative, Message, Sender),
16  select(Performative, Message, Sender),!.

18 handle_msgs.

20 select(Performative, Message, Sender) :-
 bound(Performative),
22  bound(Message),
 unpack(Message,TermMsg),
24  handle(Performative, TermMsg, Sender).

26 select(_,_,_).

28 handle('INFORM',reserve_price(Pr),Sender):-
 assert(present_bid(Pr)).
```

```
30
handle('INFORM',start(Time_H,Time_Min,Num),Sender) :-
32  assert(start(Time_H,Time_Min)), assert(num_winners(Num)),!.

34 handle('INFORM',present_bid(Bid),Sender) :-
   change_bid(Bid),
36  retract(i_win(_)),assert(i_win(false)),!.

38 handle('INFORM',you_win(Num,Bid),Sender) :-
   Num = 1,
40  retract(num_winners(_)),assert(num_winners(Num)),
    retract(i_win(_)),assert(i_win(true)),
42  blocking_receive(Performative, Message, OSender),
    change_bid(Bid),!.
44
handle('INFORM',you_win(Num,Bid),Sender) :-
46  retract(num_winners(_)),assert(num_winners(Num)),
    retract(i_win(_)),assert(i_win(false)),
48  blocking_receive(Performative, Message, OSender),
    change_bid(Bid),!.
50
handle('INFORM',last_win(Num,Bid),Sender) :-
52  retract(num_winners(_)),assert(num_winners(Num)),
    retract(i_win(_)),assert(i_win(true)),
54  change_bid(Bid),!.

56 handle('INFORM',real_winner(Winner,Bid),Sender) :-
   my_name(Winner),retract(i_win(_)),assert(i_win(true)).
58
handle('INFORM',auct_end,Sender) :-
60  assert(auct_end).

62 handle(_,_,_).


64
change_bid(Bid) :-
66    retract(present_bid(_)),assert(present_bid(Bid)).

68 get_w_bid :-
     blocking_receive(Performative, Message, Sender), bound(Performative),
70   bound(Message),unpack(Message,TermMsg),get_w_aux(TermMsg),!.

72 get_w_aux(present_bid(Bid)) :- change_bid(Bid).


74
offering :- auct_end.
76
```

```
offering :-
 start(_,_),i_win(true),!.

offering :-
 start(_,_), eval_offer(New_bid), offer(New_bid),!.

offering.

eval_offer(New_bid) :-
 num_winners(N), N =\= 1, present_bid(Present_bid), object_value(Value),
 Present_bid =< Value - 1, New_bid is Present_bid + 1,!.

eval_offer(New_bid) :-
 New_bid = no.


offer(no):-!.

offer(Bid) :-
 pack(offer(Bid),Msg),
 send('PROPOSE',Msg,"auct_eng_r@Vento:1099/JADE").
```

Bibliography

[AMMM02] R. Albertoni, M. Martelli, V. Mascardi, and S. Miglia. Specifica, implementazione ed esecuzione di un prototipo di sistema multi–agente in $DCaseLP$. In *Proc. of WOA 2002*, Milano, Italy, 2002. Pitagora editrice, Bologna.

[ASZ91] D. Atkins, W. Stallings, and P. Zimmermann. Pgp message exchange formats. *RFC*, 1991.

[BCTR04] F. Bellifemine, G. Caire, T. Trucco, and G. Rimassa. *JADE PROGRAMMERS GUIDE*. Tilab (Telecom Italia Lab), 2004.

[BDM+99] M. Bozzano, G. Delzanno, M. Martelli, V. Mascardi, and F. Zini. Multi–agent systems development as a software engineering enterprise. In G.Gupta, editor, *Proc. of First International Workshop on Practical Aspects of Declarative Languages (PADL'99)*, pages 46–60, San Antonio, Texas, 1999. Springer-Verlag.

[Bin92] K. Binmore. *Fun and games*. D.C. Heath and Company, 1992.

[BMO01] B.Bauer, J. P. Mller, and J. Odell. chapter Agent UML: A Formalism for Specifying Multiagent Interaction. Springer-Verlag, 2001.

[CH04] L. Cabral and A. Hortacsu. The dynamics of seller reputation: Theory and evidence from $eBay$. *NBER Working Paper*, 2004.

[Del97] G. Delzanno. *Logic & Object-Oriented Programming In Linear Logic*. PhD thesis, University of Pisa, 1997. ftp://ftp.disi.unige.it/person/DelzannoG/papers/thesis.ps.gz.

[DKM+99] P. Dart, E. Kazmierczak, M. Martelli, V. Mascardi, L. Sterling, V.S. Subrahmanian, and F. Zini. Combining logical agents with rapid prototyping for engineering distributed applications. In S. Tilley and

J. Verner, editors, *Proceedings of the Nineth International Conference of Software Technology and Engineering (STEP'99)*, pages 40–49, Pittsburgh, PA, 1999. IEEE Computer Society Press.

[DOR01] E. Denti, A. Omicini, and A. Ricci. tuprolog: A light-weight prolog for internet applications and infrastructures. In I. V. Ramakrishnan, editor, *Proc. of the 3rd International Symposium on Practical Aspects of Declarative Programming (PADL01)*, pages 184–198, Las Vegas, NV, U.S.A., 2001. Springer–Verlag. Home Page: http://lia.deis.unibo.it/research/tuprolog/.

[FG98] J. Ferber and O. Gutknecht. A meta-model for the analisys of organizations in multi-agent systems. In *Proceeding 3rd International Conference on Multi-Agent Systems (ICMAS-98)*, pages 128–135, Paris, France, 1998.

[FH] E. Friedman-Hill. *Jess^{TM}, the rule engine for the java platform*. Home Page: http://herzberg.ca.sandia.gov/jess/index.shtml.

[FIP] FIPA (The Foundation for Intelligent Physical Agents). *AUML. Agent–based Unified Modeling Language*. Home Page: http://www.auml.org/.

[Fis79] Peter Fishburn. *Utility Theory for Decision Making*. Krieger, Huntington (NY),, 1979.

[GLS67] J.H. Griesmer, R.E. Levitan, and M. Shubik. Towards a study of bidding process: part iv. *Naval research logistics quarterly*, 14:415–33, 1967.

[GMP02] F. Giunchiglia, J. Mylopoulos, and A. Perini. The tropos software development methodology: processes, models and diagrams. In *AAMAS02 workshop on Agent Oriented Software Engineering (AOSE-2002)*, pages 63 – 74, 2002.

[Gun05] I. Gungui. Integrating logical agents into $DCaseLP$. Master's thesis, DISI, University of Genoa, Italy, 2005.

[Har68] J. Harsanyi. Games with incomplete information played by bayesian players. *Management Science*, 14, 1967–1968.

[Hug02] M-P. Huget. Model checking agent uml protocol diagrams. Technical report, CS Department, University of Liverpool, UK, 2002.

[Koh90] E. Kohlberg. Refinement of the nash equilibrium: the main ideas. In T. Ichiishi A. Neyman Y. Tauman, editor, *Game theory and applications*. Academic Press, 1990.

[Kre88] David Mark Kreps. *Notes on the Theory of Choice*. Princeton University Press, 1988.

[LM95] T. Finin Y. Labrou and J. Mayfield. Kqml as an agent communication language. In J. Bradshaw, editor, *Software Agents*, pages 265–284. MIT Press, 1995.

[Mae94] Pattie Maes. Agents that reduce work and information overload. *Communications of the ACM*, 37, 1994.

[Mar99] S. Marini. Specifica di sistemi multi-agente eterogenei. Master's thesis, DISI, University of Genoa, Italy, 1999.

[Mas02] V. Mascardi. *Logic-Based Specification Environments for Multi-Agent Systems*. PhD thesis, University of Genova, 2002. ftp://ftp.disi.unige.it/person/MascardiV/Tesi/mythesis.ps.gz.

[Mig02] S. Miglia. Specifica ed implementazione di ruoli e protocolli dinterazione per agenti in $DCaseLP$. Master's thesis, DISI, University of Genoa, Italy, 2002.

[MMMZ00] S. Marini, M. Martelli, V. Mascardi, and F. Zini. Hemasl: A flexible language to specify heterogeneous agents. In A. Corradi A. Omicini and A. Poggi, editors, *Proc. of WOA '00. Dagli Oggetti Agli Agenti*, pages 76–81, Parma, Italy, 2000. Pitagora editrice, Bologna.

[MMZ99] M. Martelli, V. Mascardi, and F. Zini. Specification and simulation of multi–agent systems in $CaseLP4$. In *Proc. of Appia-Gulp-Prode*, L'Aquila, Italy, 1999.

[MW82] P. Milgrom and R.J. Weber. A theory on auctions and competitive bidding. *Econometrica*, 50:1089–1122, 1982.

[Mye81] R.B. Myerson. Optimal auction design. *Mathematics of Operations Research*, 6:58–73, 1981.

[Mye91] R.B. Myerson. *Game Theory. Analysis of conflict*. Harvard University Press, 1991.

[NAS] NASAs Johnson Space Center. *CLIPS: A Tool for Building Expert Systems*. Home Page: http://www.ghg.net/clips/CLIPS.html.

[Nas51] J. Nash. Non–cooperative games. *Annals of Mathematics*, 54:286–295, 1951.

[NM44] J. Von Neumann and O. Morgenstern. *The theory of games and economic behaviour*. Westview Press, 1944.

[Rod01] J. A. Rodriguez. On the design and construction of agent-mediated electronic institutions. *IIIA Monographs*, 14, 2001.

[RS81] J.G. Riley and W.F. Samuelson. Optimal auctions. *American economic review*, 71:381–92, 1981.

[RSGJ03] S. D. Ramchurn, C. Sierra, L. Godo, and N. R. Jennigs. A computational trust model for multiagent interactions based on confidence and reputation. In *Proceedings of International Workshop on Deception, Trust, and Fraud, AAMAS03*, pages 69 – 75, 2003.

[SA03] F. Stolzenburg and T. Arai. From the specification of multi–agent systems by statecharts to their formal analysis by model–checking: Towards safety-critical applications. In J. Muller M.Schillo M.Klusch and H.Tianfield, editors, *Proc. of the First German Conference on Multiagent System Technologies*, pages 131–143. Springer-Verlag, 2003.

[Sab03] J. Sabater. Trust and reputation for agent societies. In *IIIA Monographs, CSIC: Barcelona*, volume 20, 2003.

[SIC] SICS. (SWEDISH INSTITUTE OF COMPUTER SCIENCE). *SICStus Prolog*. Home Page: http://www.sics.se/isl/sicstuswww/site/index.html.

[Sie04] C. Sierra. Agent-mediated electronic commerce. *Autonomous Agents and Multi-Agent Sytems*, 9:285–301, 2004.

[Sun] Sun Microsystems. *Java, trademark of Sun Microsystems*. Home Page: http://www.sun.com/.

[Til] Tilab (Telecom Italia Lab). *Java Agent DEvelopment Framework, an Open Source platform for peer-to-peer agent based applications*. Home Page: http://jade.tilab.com/.

[Vic61] W. Vickrey. Counterspeculation, auctions and competitive sealed tenders. *Journal of Finance*, 16:8–37, 1961.

[Vic62] W. Vickrey. Auction and bidding games. In *Recent advances in Game Theory*, pages 15–27. Princeton University Conference, 1962.

[Wil69] R. Wilson. Competitive bidding with disparate information. *Management Science*, 15:446–48, 1969.

[WJK00] M. Wooldridge, N. R. Jennings, and D. Kinny. The $Gaia$ methodology for agent–oriented analysis and design. *J. of Auton. Agents and Multi-agent Syst.*, 3(3):285–312, 2000.