TECHNICAL UNIVERSITY OF CRETE
DEPARTMENT OF PRODUCTION ENGINEERING AND MANAGEMENT
DECISION SUPPORT SYSTEMS LABORATORY

Development of a multi-agent system for the support of group decisions utilizing argumentation and multicriteria methods

Ανάπτυξη ενός συστήματος ευφυών πρακτόρων υποστήριξης της λήψης ομαδικών αποφάσεων και διαπραγμάτευσης με επιχειρήματα, με χρήση πολυκριτήριων μεθόδων

Ph.D. Thesis

by

Konstantinos-Dimitrios Tzoannopoulos

Supervising Professor: Nikolaos F. Matsatsinis
Professor, Tecnhical University of Crete

Chania, 2011

This page intentionally left blank.

TECHNICAL UNIVERSITY OF CRETE
DEPARTMENT OF PRODUCTION ENGINEERING AND MANAGEMENT
DECISION SUPPORT SYSTEMS LABORATORY

Development of a multi-agent system for the support of group decisions utilizing argumentation and multicriteria methods

Ανάπτυξη ενός συστήματος ευφυών πρακτόρων υποστήριξης της λήψης ομαδικών αποφάσεων και διαπραγμάτευσης με επιχειρήματα, με χρήση πολυκριτήριων μεθόδων

Ph.D. Thesis

by

Konstantinos-Dimitrios Tzoannopoulos

Supervising Professor: Nikolaos F. Matsatsinis
Professor, Technical University of Crete

Chania, December 2011

(Signature)

.....

KONSTANTINOS-DIMITRIOS TZOANNOPOULOS

The author would like to thank the staff of the Decision Support Systems Laboratory and especially Mrs. Lia Krasadaki,M.Sc. Production Engineer, for their assistance in this Ph.D. thesis.

Furthermore, the author would like to thank his family for all their support through the process of his doctorate studies. Special thanks are extended to software engineer Konstantinos Bokaris for lending his invaluable advice and expertise for the development and debugging of the software that accompanies this thesis.

Abstract

Group decision making is one of the most important and frequently encountered processes within companies and organizations, both in the public and private sectors (Turban 1988). The understanding, analysis and support of this process is difficult, due to the ill-structured, dynamic environment and the presence of multiple Decision Makers (DMs); each DM has his or her own perceptions and views on how the problem should be handled and which decision should be made (Jelassi et al. 1990).

Thanks to the developments in multicriteria decision making methodologies and the increasing popularity of computerized MCDM methods, scientists and professionals have been provided with a set of tools whose usage can be advantageous in solving problems with multiple criteria. However, it is evident that the effectiveness of such procedures when used by multiple DMs remains to be proven. This necessitates the use of practical aggregation methods to extend the existing MCDM methodologies, as well as the computing methodologies, to support group decision problems (Iz and Krajewski 1992).

The use of Group Decision Support Systems (GDSSs) is crucial when multiple persons are involved in the decision making process, since each DM has his or her own perceptions of the context and the decision problem at hand. In environments of this kind, the occurrence of conflicts among the members of the decision-making group is frequent. This conflict is referred to as *interpersonal conflict* (Bogetoft and Pruzan 1991). Factors that contribute to the occurrence of interpersonal conflicts include different values and objectives, different criteria and preference relations, lack of communication support among the members of the decision-making group etc. Noori (1995) recognizes that, from a practical point of view, conflicting objectives among the members of a group often exist due to interpersonal differences and goal incongruities.

In coping with interpersonal conflicts, the aim is to achieve consensus among the DMs; in such problems, Multi-Criteria Decision Aid (MCDA) methods may be a useful tool. As argued by Bui and Jarke (1986), MCDM/MCDA methods provide an elegant framework for three important GDSS tasks: (a) representing multiple viewpoints of a problem, (b) aggregating the preferences of multiple DMs according to various group norms and (c) organizing the decision process. The framework offered by MCDM is simple but structured, while the simplicity of its outputs makes communicating, coordinating and aggregating individual analyses in the group decision making process easier. Jarke (1986) states that MCDM methods can serve as formal tools for preference surfacing and aggregation, as well as negotiation and mediation, in both cooperative and non-cooperative decision situations. Thus, the multiple criteria process of a GDSS is a key aspect of the system, as it provides a structured and integrated framework for the assessment of alternatives and criteria and for solution compromise.

Multiple agent (multi-agent) systems have become a valuable tool in the field of group decision making and, recently, their application has been extended in the sector of group decision making using multicriteria decision analysis techniques. For the decision making process in a multi-agent context, numerous techniques and methodologies have been proposed and implemented over the years, providing various solutions and approaches to the problem of group decision making.

The term "*interaction*" rather than "*argumentation-based negotiation*" has been chosen, because, although this methodology did actually begin as *argumentation-based negotiation* (Sycara 1989b, Parsons et al. 1998), it has now branched out and evolved into a completely unique type of multi-agent interaction in its own right, overcoming and surpassing the limitations of game-theoretic negotiation. It appears to be a very viable and promising methodology, because of the close and efficient approximation of the procedure used by human decision makers it achieves.

This thesis proposes a Group Decision Support methodology and software system that attempts to support a group of decision makers bestowed with the solution of a choice problem (i.e. a problem of the problematic α), beginning from a set of individual ordinal rankings and using a combination of a heuristic algorithm and an argumentation protocol for the building of a consensus among the decision makers. The heuristic algorithm serves as a method of accelerating the argumentation-based negotiation on the alternatives.

This thesis is part of the 03ED375 research project, implemented within the framework of the "Reinforcement Programme of Human Research Manpower" (PENED) and co-financed by National and Community Funds (75% from E.U.-European Social Fund and 25% from the Greek Ministry of Development-General Secretariat of Research and Technology).

Keywords: multi-criteria decision analysis, UTASTAR, argumentation, argumentation-based negotiation, group decision support, consensus

This page intentionally left blank.

Group decision making is one of the most important and frequently encountered processes within companies and organizations, both in the public and private sectors (Turban 1988). The understanding, analysis and support of this process is difficult, due to the ill-structured, dynamic environment and the presence of multiple Decision Makers (DMs); each DM has his or her own perceptions and views on how the problem should be handled and which decision should be made (Jelassi et al. 1990).

Thanks to the developments in multicriteria decision making methodologies and the increasing popularity of computerized MCDM methods, scientists and professionals have been provided with a set of tools whose usage can be advantageous in solving problems with multiple criteria. However, it is evident that the effectiveness of such procedures when used by multiple DMs remains to be proven. This necessitates the use of practical aggregation methods to extend the existing MCDM methodologies, as well as the computing methodologies, to support group decision problems (Iz and Krajewski 1992).

The use of Group Decision Support Systems (GDSSs) is crucial when multiple persons are involved in the decision making process, since each DM has his or her own perceptions of the context and the decision problem at hand. In environments of this kind, the occurrence of conflicts among the members of the decision-making group is frequent. This conflict is referred to as *interpersonal conflict* (Bogetoft and Pruzan 1991). Factors that contribute to the occurrence of interpersonal conflicts include different values and objectives, different criteria and preference relations, lack of communication support among the members of the decision-making group etc. Noori (1995) recognizes that, from a practical point of view, conflicting objectives among the members of a group often exist due to interpersonal differences and goal incongruities.

In coping with interpersonal conflicts, the aim is to achieve consensus among the DMs; in such problems, Multi-Criteria Decision Aid (MCDA) methods may be a useful tool. As argued by Bui and Jarke (1986), MCDM/MCDA methods provide an elegant framework for three important GDSS tasks: (a) representing multiple viewpoints of a problem, (b) aggregating the preferences of multiple DMs according to various group norms and (c) organizing the decision process. The framework offered by MCDM is simple but structured, while the simplicity of its outputs makes communicating, coordinating and aggregating individual analyses in the group decision making process easier. Jarke (1986) states that MCDM methods can serve as formal tools for preference surfacing and aggregation, as well as negotiation and mediation, in both cooperative and non-cooperative decision situations. Thus, the multiple criteria process of a GDSS is a key aspect of the system, as it provides a structured and integrated framework for the assessment of alternatives and criteria and for solution compromise.

Multiple agent (multi-agent) systems have become a valuable tool in the field of group decision making and, recently, their application has been extended in the sector of group decision making using multicriteria decision analysis techniques. For the decision making process in a multi-agent context, numerous techniques and methodologies have been

proposed and implemented over the years, providing various solutions and approaches to the problem of group decision making.

The term "*interaction*" rather than "*argumentation-based negotiation*" has been chosen, because, although this methodology did actually begin as *argumentation-based negotiation* (Sycara 1989b, Parsons et al. 1998), it has now branched out and evolved into a completely unique type of multi-agent interaction in its own right, overcoming and surpassing the limitations of game-theoretic negotiation. It appears to be a very viable and promising methodology, because of the close and efficient approximation of the procedure used by human decision makers it achieves.

This thesis proposes a Group Decision Support methodology and software system that attempts to support a group of decision makers bestowed with the solution of a choice problem (i.e. a problem of the problematic α), beginning from a set of individual ordinal rankings and using a combination of a heuristic algorithm and an argumentation protocol for the building of a consensus among the decision makers. The heuristic algorithm serves as a method of accelerating the argumentation-based negotiation on the alternatives.

This thesis is part of the 03ED375 research project, implemented within the framework of the "Reinforcement Programme of Human Research Manpower" (PENED) and co-financed by National and Community Funds (75% from E.U.-European Social Fund and 25% from the Greek Ministry of Development-General Secretariat of Research and Technology).

Keywords: multi-criteria decision analysis, argumentation, argumentation-based negotiation, group decision support

This page intentionally left blank.

TABLE OF CONTENTS

Chapter I *INTRODUCTION*

This chapter is an introduction to multicriteria group decision support using argumentation. In the beginning, a short historical review is presented; the basic notions (multicriteria group decision support, argumentation) are set forth. Finally, the scope of this thesis is presented.

I.1 Historical review

Group decision making is one of the most important and frequently encountered processes within companies and organizations, both in the public and private sectors [Turban, (1988)]. The understanding, analysis and support of this process is difficult, due to the ill-structured, dynamic environment and the presence of multiple Decision Makers (DMs); each DM has his or her own perceptions and views on how the problem should be handled and which decision should be made (Jelassi et al 1990).

Thanks to the developments in multicriteria decision making methodologies and the increasing popularity of computerized MCDM methods, scientists and professionals have been provided with a set of tools whose usage can be advantageous in solving problems with multiple criteria. However, it is evident that the effectiveness of such procedures when used by multiple DMs remains to be proven. This necessitates the use of practical aggregation methods to extend the existing MCDM methodologies, as well as the computing methodologies, to support group decision problems (Iz and Krajewski 1992).

The use of Group Decision Support Systems (GDSSs) is crucial when multiple persons are involved in the decision making process, since each DM has his or her own perceptions of the context and the decision problem at hand. In environments of this kind, the occurrence of conflicts among the members of the decision-making group is frequent. This conflict is referred to as *interpersonal conflict* (Bogetoft and Pruzan 1991). Factors that contribute to the occurrence of interpersonal conflicts include different values and objectives, different criteria and preference relations, lack of communication support among the members of the decision-making group etc. Noori (1995) recognizes that, from a practical point of view, conflicting objectives among the members of a group often exist due to interpersonal differences and goal incongruities.

In coping with interpersonal conflicts, the aim is to achieve consensus among the DMs; in such problems, Multi-Criteria Decision Aid (MCDA) methods may be a useful tool. As argued by Bui and Jarke (1986), MCDM/MCDA methods provide an elegant framework for three important GDSS tasks: (a) representing multiple viewpoints of a problem, (b) aggregating the preferences of multiple DMs according to various group

norms and (c) organizing the decision process. The framework offered by MCDM is simple but structured, while the simplicity of its outputs makes communicating, coordinating and aggregating individual analyses in the group decision making process. Jarke (1986) states that MCDM methods can serve as formal tools for preference surfacing and aggregation, as well as negotiation and mediation, in both cooperative and non-cooperative decision situations. Thus, the multiple criteria process of a GDSS is a key aspect of the system, as it provides a structured and integrated framework for the assessment of alternatives and criteria and for solution compromise.

Multiple agent (multi-agent) systems have become a valuable tool in the field of group decision making and, recently, their application has been extended in the sector of group decision making using multicriteria decision analysis techniques. For the decision making process in a multi-agent context, numerous techniques and methodologies have been proposed and implemented over the years, providing various solutions and approaches to the problem of group decision making.

The term "*interaction*" rather than "*argumentation-based negotiation*" has been chosen, because, although this methodology did actually begin as *argumentation-based negotiation* (Sycara, 1989b, Parsons et al. 1998), it has now branched out and evolved into a completely unique type of multi-agent interaction in its own right, overcoming and surpassing the limitations of game-theoretic negotiation. It appears to be a very viable and promising methodology, because of the close and efficient approximation of the procedure used by human decision makers it achieves.

In this thesis, a multi-criteria Group Decision Support System (GDSS) is presented, which helps a group of Decision Makers (DMs) solve a *choice* problem, i.e. a problem in which the DMs are presented with a number of alternatives and have to choose the one that seems to be optimal for them (problematic α). To achieve this, the software aggregates and the preferences of the individual (DMs); these preferences are formulated into individual ordinal rankings of the alternatives. Then, the GDSS attempts to build a consensus among the DMs by applying a combination of a heuristic algorithm – more specifically, the Negotiable Alternative (NAI) algorithm [Bui 1985, Bui and Shakun 1987) – , which has been adapted to the needs of this particular methodology and an argumentation protocol that enables the DMs (or agents that represent them) to negotiate using arguments *for* and *against* each alternative, in order to eventually propose the best commonly accepted alternative. The method chosen for the disaggregation of the DMs' preferences is the UTASTAR (Siskos and Yannacopoulos 1985, Siskos et al. 2005). Following the calculation of the DMs' individual ordinal rankings is the NAI algorithm stage. Finally, the argumentation protocol is based on a framework presented by Amgoud *et al* [Amgoud *et al* (2005)] and provides automatic argument generation and assessment-evaluation by comparing the strengths of the arguments. The strength of each argument is determined by already given or calculated data (criterion weight, performance of the alternative on the specific criterion).

I.2 Acknowledgements

This thesis is part of the 03ED375 research project, implemented within the framework of the "Reinforcement Programme of Human Research Manpower" (PENED) and co-

financed by National and Community Funds (75% from E.U.-European Social Fund and 25% from the Greek Ministry of Development-General Secretariat of Research and Technology). The author gratefully acknowledges the cooperation and assistance of the project's scientific advisor Prof. Nikos F. Matsatsinis, fellow PENED researchers Pavlos Delias, Klio Lakiotaki and Stelios Tsafarakis, and the assistance of the staff of the Technical University of Crete's Decision Support Systems Laboratory (ERGASYA). Also, the author wishes to thank software engineer Konstantinos Bokaris for lending his expertise in the Java programming language and his invaluable advice for the development and debugging of the accompanying software.

I.3 Literature Review

I.3.1 Previous negotiation/argumentation systems and protocols

In the field of group decision making, many researchers have presented decision methodologies; comprehensive lists of such methodologies can be found in Hwang and Ling (1987) and Matsatsinis and Samaras (2001).

In this chapter, an attempt will be made to document and present, in a manner as concise and complete as possible, the most important developments and advances in the field of multicriteria group decision making, with emphasis given to systems implementing multi-agent and multi-user argumentation.

NEGO, presented by Kersten (1985), is a two-stage interactive procedure of individual proposal formulation and negotiation that leads to compromise based on the generalized theory of negotiations' formulation developed by Kersten and Szapiro (1985). The Co-oP system is one of the most well-known and documented implementations (Bui 1987, Bui and Jarke 1986): it is a GDSS for cooperative multicriteria decision making. It can be used either for the ranking of alternatives using the Analytic Hierarchy Process (AHP) method (Saaty 1980) or for selecting one, and only one, alternative using the ELECTRE method (Roy 1968).

MEDIATOR (Jarke et al. 1987) is a negotiation support system based on evolutionary systems design and database-centered implementation with many applications (Giordano et al. 1988, Shakun 1988, 1991). Kersten (1987) discusses the role that MEDIATOR and NEGO can play in negotiations. Lewandowski's SCDAS (Lewandowski 1989) is a system that can support a group of DMs working together on selecting the best alternative from a finite, given set of alternatives. Vetchera (1991) makes use of the multi-attribute utility theory to develop a general framework for group decision support combining the reduction in cognitive strain provided by individual views with feedback processes. Iz and Krajewski (1992) propose extensions in three single decision maker procedures for multicriteria problems based on interactive multiple objective linear programming (MOLP) techniques.

Carlsson et al. (1992) present *Alicia & Sebastian*, a system for formalizing consensus reaching within a set of DMs trying to find and agree upon a mutual decision. In *Alicia & Sebastian*, the AHP method is used to model the preferences of each DM. In Dyer and Forman (1992), it is argued that the AHP method (Saaty 1980) works well for group decision making, because it offers numerous benefits as a synthesizing mechanism in group decisions. In their aforementioned work, Dyer and Forman describe four ways in which the

AHP method can be applied to the common objectives context: (1) consensus, (2) voting or compromising, (3) forming the geometric means of individuals' judgements, and (4) combining results from individual models or parts of a model. JUDGES (Colson and Mareschal 1994) is a descriptive GDSS for the cooperative ranking of the alternatives.

Choi, Suh and Suh (1994) discuss the applicability and practicality of the AHP method in a GDSS for a new provincial seat selection in South Korea. Csáki et al. (1995a, 1995b) present WINGDSS, a GDSS designed to support one or more DMs from different fields but with a common interest in ranking a finite set of alternatives that are characterized by a finite set of criteria and attributes. Salo (1995) developed an interactive approach for the aggregation of the DMs' preference judgments in the context of an evolving value representation. Stanoulov (1994, 1995) presented the dichotomic matrix multiple criteria optimization (DIMCO) method, which is an outranking approach for individual and group decision making. Noori (1995) presented a conceptual design of a GDSS named NTech-GDSS, developed to guide management through the process of evaluating and adopting new technologies. Barzilai and Lootsma (1997) use the multiplicative AHP method (Lootsma 1993), a variant of the original AHP method, to reach a joint decision by incorporating the relative power of the DMs.

Obviously, the number of argumentation-based multi-agent group decision support systems cannot be overwhelmingly large; while it is true that multi-agent systems play an increasingly important role in the field of decision making, argumentation-based multi-agent decision support systems are a very recent development. Thus, it is understandable that, among multi-agent DSSs, the number of systems that make use of multicriteria decision analysis (MCDA) methods can only be relatively small. For historical reasons only, some important works shall be mentioned. The PERSUADER system by Sycara (1989a, b, 1990) was perhaps the first argumentation-based multi-agent system; it operated in the field of labour negotiation and involved three agents: an agent representing a labour union, an agent representing a company and a third agent acting as a mediator. Its task was to model the iterative exchange of proposals and counter-proposals so that the parties would reach an agreement. The negotiation involved multiple issues (wages, pensions, seniority, etc). It must be noted here that the inherent ability of argument-based multi-agent systems to handle multiple-issue problems does not make them multicriteria applications. In PERSUADER, the argumentation used a model of each agent's beliefs; these beliefs captured an agent's goals and interrelationships among them.

Aiming to broaden the scope of argumentation research and encourage the development of more ambitious computer implementations than those available at the time (e.g. the OpEd and SIBYL systems), John A. A. Sillince (1993) proposed a system, in which agents attempted to make claims by using tactical rules (e.g. fairness and commitment) and said what other claims were supported or attacked by the claims they made. The claims could be sets of claims connected by attacking and supporting links. Over the course of the debate, a shared argument map was generated; this map was controlled by a set of strategic rules; the purpose of these rules was to keep the location of focus within the argument map under control. An aspect of his work that differentiated it from other research works of the time was the lack of a requirement for truth propagation and consistency maintenance. The arguments' strength was calculated in terms of a number of structural constraints by means of an evaluation function. That way, invulnerability to attack due to self-inconsistency was just one of several criteria (other criteria included constructiveness, relevance and familiarity). The arguments themselves were mappings

from source to target domains and were constructed by three knowledge sources: quasi-logic, value transfer and emotional appeal.

Dung (1995) contributed a seminal paper, in which he studied the fundamental mechanism used by humans in their argumentation, in order to implement it on computers; he developed an argumentation theory, with the acceptability of arguments being the central notion. He approached argumentation as a special form of logic programming, where negation is failure and introduced a general logic programming-based method for the generation of meta-interpreters for argumentation systems, a method similar to the compiler-compiler concept in conventional programming. The way of looking at arguments he proposed was more abstract: rather than looking at the internal structure of individual arguments, he suggested that one looks at the *overall structure* of the argument. He modeled such an *abstract argument system A* as a pair:

$$A \equiv <X, \rightarrow>,$$

where

- X is a set of arguments (*what* the members of X are is irrelevant);
- $\rightarrow \subseteq X \times X$ is a binary relation on the set of arguments, representing the notion of *attack*.

Karacapilidis et al. (1996) presented an argumentation-based framework written in Java, which supported defeasible and qualitative reasoning in a multi-agent context. The logic applied was interval-based, combined with an inference engine which served the purposes of refining the agents' knowledge, checking consistency and concluding the issue. Like Sycara's PERSUADER, this framework supports multiple-issue argumentation. Taking the lead from Dung (1995), Verheij (1996) proposed a model for the argumentation stages; each stage was characterized by the arguments taken into account and their status (defeated or undefeated). His approach provided good understanding of the argumentation process, because sequences of stages could be interpreted as lines of argumentation – from this stage approach two new types of extensions emerged, with their definitions formalizing the idea that as many arguments were being taken into account as possible. He also concluded that the argumentation stage approach, which is a generalization of the admissible sets approach, provides better insight into the procedural nature of dialectical argumentation than the admissible sets approach.

Simon Parsons, together with Carles Sierra and Nick R. Jennings (Parsons et al. 1997), aiming to provide a better alternative to the usual agent architectures, which were somewhat *ad hoc* in nature, proposed an agent design approach based on multi-context systems, which are a framework allowing the definition and interrelation of distinct theoretical components, and argumentation, in order to allow the development of agent architectures equipped with a formal model in logic and a direct link between this model and its implementation. As an example of this approach, they presented a case study of the strong realist Belief-Desire-Intention model. Chris Reed and Derek Long (Reed and Long 1997) presented an ordering technique designed to enhance coherence in a persuasive discourse, so that the resulting functionality could generate plans closely resembling structures found in natural argument.

Karacapilidis and Papadias' proposal (Karacapilidis and Papadias 1997) was for a group decision and argumentation support system for cooperative and non-cooperative discourses, which provided agents with means of expressing and weighing their individual arguments and preferences; the aim was the selection of a certain choice. This system also supports defeasible and qualitative reasoning in the presence of ill-structured information. The entire argumentation process is performed through a set of discourse acts, which call a variety of procedures to propagate information in the corresponding discussion graph. Also, Karacapilidis, in collaboration with Brigitte Trousse (Karacapilidis and Trousse 1997), presented an argumentation system for cooperative design on the web, whose features were almost identical to the one mentioned above – perhaps the system presented by Karacapilidis and Trousse was the predecessor of the one proposed in Karacapilidis and Papadias, (1997). In Karacapilidis and Trousse (1997), a report was made on the integration of Case-Based Reasoning techniques, used for the resolution of current design issues through the consideration of previous similar situations, and the specification of similarity measures among the various argumentation items, with the estimation of the variations among the participating designers' opinions being the goal.

In 1998, Karacapilidis and Papadias (1998) developed HERMES, a web-based (thus providing inexpensive access to a broad public) GDSS written in Java and employing multi-agent argumentation, capable of handling incomplete, qualitative and inconsistent information, equipped with mechanisms for weighing arguments. HERMES organized the existing knowledge in a discussion graph consisting of issues, alternatives, positions and preference relations. It could be used for distributed, synchronous or asynchronous collaboration, overcoming the requirement for the agents to be in the same place and working at the same time. Argumentation was carried out through a set of discourse acts triggering appropriate procedures for the propagation of information in the graph. Although HERMES was capable of handling multiple issues, it did not incorporate multicriteria decision theory methodologies.

Kraus, Sycara and Evenchik (1998) presented argumentation as an iterative process for a multi-agent environment where self-motivated agents strive to persuade each other and bring about a change in intentions; argumentation was dealt with as a mechanism for the achievement of cooperation and agreements. Through the usage of categories identified from human multi-agent negotiation, the utilization of logic for the formulation and evaluation of arguments was demonstrated. Furthermore, a general Automated Negotiation Agent, based on their logical model, was presented. This system enabled the user to analyse and explore different negotiation and argumentation methods in a non-cooperative environment without a centralized coordination mechanism. Another argumentation framework was proposed by Parsons, Sierra and Jennings (1998): it provided a formal model of argumentation-based reasoning and negotiation and detailed a design philosophy that ensured a clear link between the formal model and its practical instantiation. Other dialogue frameworks for multi-agent argumentation have been presented by Reed (1998) and Sierra et al. (1998): the former paper offered a formal characterization which clearly distinguished persuasion from negotiation and also introduced three other dialogue types – then, Reed proceeded to set all five types in a coherent framework; the latter was basically a negotiation/argumentation framework, i.e. the agents exchanged proposals and counter-proposals, backed by arguments. The argumentation in the framework presented by Sierra et al. (1998) was persuasive, because the exchanges were able to change the mental state of the agents involved.

Boella, Hulstijn and van der Torre (2006) introduced a logic of abstract argumentation which captures Dung's theory of abstract argumentation, based on connectives for attack and defense, and extended it to a modal logic of abstract argumentation that generalizes Dung's theory and defines variants of it. They also use this logic to relate Dung's theory of abstract argumentation to more familiar and traditional conditional and comparative formalisms, illustrating ways of reasoning about arguments in meta-argumentation.

Wooldridge, McBurney and Parsons (2006) proposed an approach to the formalization of argument systems; taking the view that arguments and dialogues are inherently *meta-logical* (and the view that any proper formalization of arguments must embrace this aspect of their nature) as their starting point, they developed a formalization of arguments using a hierarchical first-order meta-logic in which statements in successively higher tiers of the argumentation hierarchy refer to statements further down the hierarchy. This provides a clean formal separation between object-level statements, arguments *about* these object level statements, and statements about arguments.

I.3.2 Multicriteria protocols, applications and implementations

The first steps towards the integration of MCDA methods in multi-agent DSSs were the Tête-à-Tête, described in Maes et al. (1999), along with Logical Decisions for Windows, which was produced by Logical Decisions, Inc. and also described in Guttman and Maes (1998). Tête-à-Tête used multi-attribute utility theory. Its further development continues nowadays by the Frictionless Commerce company, which was co-founded by Robert Guttman and Alexandros Moukas. It was quite an innovation, because, whereas the agents in all the other e-commerce systems of the time negotiated on the product's *price*, Tête-à-Tête's agents negotiated on a *multitude* of product attributes, such as warranties, features etc. Furthermore, it was not just a system with argumentation over multiple *issues*, but went further, employing techniques and ideas taken from multi-attribute utility theory.

Matsatsinis et al. (1999) presented an agent-based system which implemented a consumer-based methodology for product penetration strategy selection in real world situations. In this system, the agents were simultaneously considered according to a functional and a structural level. In the functional level, the system had three agent types: task agents, information agents and interface agents assuming the fulfilment of the task through cooperation, information gathering tasks and mediation between the users' agents and the artificial ones respectively. In the structural level, there were elementary agents based on a generic reusable architecture, as well as complex agents considered as an agent organization created dynamically in a hierarchical way. Karacapilidis and Moraïtis (2001) developed a web-based multicriteria e-commerce system incorporating the use of argumentation. In this system, salesmen and customers delegate their roles to the agents. The messages passed between these agents can completely contain the associated parties' points of view towards a market transaction. Describing things in a more specific manner, an offer request consists of a list of the product's attributes the customer wants to know about, a partial order of their importance and the constraints imposed. From the salesman's side, an offer proposal can be made according to information conveyed in the customer's offer request. This system also has a few advanced features; for instance, the agents stay in the market permanently, thus learning from it. The agents can act proactively to initiate a

transaction. More importantly, an interactive multicriteria decision-making tool has been integrated into the system, enabling the agent-buyer to perform a comparative evaluation of the proposals semi-autonomously. More recently, Moraïtis and Tsoukiàs (2003) demonstrated how argumentation can be used in the decision aiding process, implying the use of multiple criteria. Additional work was done by Dimopoulos et al. (2004) on the subject of argumentation-based modeling of decision aid for autonomous agents; this model is presented as amenable to automation and can be embedded in autonomous agents in order to enable them to support a decision maker or completely substitute him. Decision aiding is treated as a defeasible reasoning process.

While not a multicriteria system in itself at the moment, the work on argument-based negotiation presented by Kakas and Moraïtis (2006) deserves mentioning, because the researchers have clearly stated their intention to expand their proposed argument-based negotiation protocol to use multicriteria techniques from decision theory in the evaluation, on behalf of the agents, of the exchanged offers and counter-offers. In the already developed protocol, offers by the negotiating parties are linked to different arguments that the agents can build according to their individual argumentation strategies. The proposed protocol can take the agents' different roles and context of interaction into account if necessary, i.e. when the arguments' strength depends on these factors. The agents can adapt their negotiation strategies and offers as necessary during the course of the negotiation. Additionally, this system uses abduction, enabling the agents to find negotiating conditions to support an argument for an offer, extending the negotiation object in order to assist the reaching of an agreement.

An important ongoing project is called HealthAgents (Lluch-Ariet et al. 2008), González-Vélez, H. et al. 2006, Arús et al. 2006). It is a combination of a distributed DSS (d-DSS) and a distributed data warehouse (d-DWH). It will provide advanced distributed data mining functionalities for the analysis and interpretation of brain tumour data. The system's d-DWH will include the world's largest network of interconnected databases (Data Marts) of clinical, histological and molecular phenotype data of brain tumour patients. The d-DSS's mission will be to facilitate evidence-based clinical decision making using MR and genetic-based tumour classifications and will also include new criteria from the automated analysis of each local database. The goal of the HealthAgents project is to create a user-friendly web-based d-DSS for the accurate diagnosis and prognosis of brain tumours. Particular attention will be paid to child brain tumours, whose aetiology and social impact differ to those of adult brain tumours. The researchers involved had also set up an informative website for their project at http://www.healthagents.net; now that website is defunct and the entire project in its stable version has been released to the user and developer community as Free/Libre and Open Source Software at the SourceForge repository (https://sourceforge.net/projects/healthagents/).

A year earlier, in 2005, Amgoud et al. (2005) presented a general formal argumentation framework for multi-criteria decision making by a group of DMs. In their work, autonomous agents engage in a dialogue looking for a common agreement on a collecttive choice. The setting of this framework has three main components: the agents, their reasoning capabilities and a protocol. The agents are supposed to hold certain beliefs about their environment and the other agents, along with their own individual goals. The beliefs have a degree of certainty (i.e. they are more or less certain) and the goals may or may not have equal priorities. The agents are also supposed to be able to make decisions, revise their beliefs and support their points of view using arguments about the positive and

negative aspects of each decision (i.e. of each alternative). A multicriteria decision problem was formalized within a logical argumentation system and an illustrative example was provided. Amgoud, in collaboration with other researchers, has also proceeded to become a prolific researcher in the field of argumentation-based decision making, with numerous articles; most of them build and improve upon existing techniques she has proposed and she has proposed argumentation frameworks and protocols both for multi-agent and single-agent contexts.

In 2010, Meir Kalech and Avi Pfeffer (2010) proposed a multi-agent model which attempts to address the problem of decision making with dynamically arriving information, i.e. with information that changes over time. In many real-world problems, there is a cost to waiting for more information, which raises the question when one should stop waiting and make the decision. Should the decision maker(s) stop and make the best decision possible or should they wait until more information arrives that will enable them to make a better decision? This model characterizes the influence of dynamic information on the utility of the decision. Using this base as a model, Kalech and Pfeffer (2010) presented an optimal algorithm that guarantees the best time to stop. However, this model is quite complex: its complexity is exponential in the number of candidates. They also presented an alternative framework in which the different candidates are sold separately. The alternative framework is analyzed formally and the way in which it leads to a range of specific heuristic algorithms is shown. Through experiments, they evaluated the optimal and the simplest heuristic algorithms, which demonstrated that the heuristic algorithm is much faster than the optimal algorithm and the utility of the winning candidate found by the heuristic algorithm is close to the optimum.

Again in 2010, Rahwan and Tohmé (2010) addressed the problem of collective decision making by a set of agents who, starting with conflicting knowledge bases (i.e. each has its own set of legitimate subjective evaluation of a set of arguments), had to reach a collective evaluation of their arguments. They analyzed an argument-wise plurality voting rule, demonstrating that it suffers from a fundamental limitation. Using a general impossibility result, they showed that this limitation is more fundamentally rooted and, finally, demonstrated a way to circumvent this impossibility result with additional domain restrictions.

I.3.3 Present and future research trends

So far, the number of multicriteria multi-agent decision support systems is rather limited. This is understandable, as MCDA methodologies have only recently begun to be integrated in multi-agent decision support systems. Most of the multi-agent decision support systems using argumentation and/or argumentation-based negotiation, including multicriteria ones, are e-commerce systems, aiding the purchase process, with one of those systems (HealthAgents) targeting the medical diagnosis and prognosis field.

The process of designing and/or developing a new product using the combination of multiple agents (which correspond to multiple users/decision makers that are represented by the agents), MCDA and argumentation or argumentation-based negotiation appears to be still largely unexplored by researchers. The development of such a system seems possible and would certainly have a place in the decision support systems market, especially if it had such desirable characteristics as web-based operation, support for

synchronous and asynchronous collaboration and interfacing with popular database systems.

I.3.4 The scope of this thesis: Combining MCDA methods, heuristics and argumentation in Group Decision Support

In this thesis, a methodological framework is proposed, and an example software application is developed and presented, that addresses the problem of supporting a group of decision makers (DMs) who want to *choose* from a set of actions (alternatives) in a collaborative multiple criteria context. In this methodology, a multi-criteria method is combined with a heuristic algorithm and an argumentation framework; the multi-criteria method used is the UTASTAR method (Siskos and Yannacopoulos 1985, Siskos et al. 2005) . The UTASTAR method is applied in order to:

1. Calculate the relative utility values *for every individual DM.*
2. Calculate each individual DM's ranking of the alternatives.

After this phase, a heuristic algorithm known as the Negotiable Alternative Identifier algorithm (Bui, 1985, Bui and Shakun 1987, Bui and Yen 1995) is employed to determine a set of negotiable alternatives, i.e. alternatives that the DMs would consider, even if they are not at the top of their individual ordinal rankings. This algorithm might seem redundant at first, but it serves as an accelerator for the argumentation process that will follow afterwards, by significantly reducing the number of alternatives upon which the agents representing the DMs will negotiate.

It is after the completion of this stage that an argumentation process commences so that *one* alternative will be eventually proposed to the group of DMs. Intuitively, one would expect that combining the individual ordinal rankings of the alternatives from most preferable to least preferable with the NAI algorithm would suffice; it would be quite easy for someone to argue that the alternative that made it to the top of the ranking list is the one that should be eventually chosen. Indeed, this thought makes sense in the context of single-user decision support. When confronted with a group of decision makers, each with his/her own personal preferential profile, the matter becomes more complicated, as each DM's ranking of the alternatives varies, often greatly, from the rankings of the other DMs. In this case, a consensus must be sought among the DMs on a *collectively acceptable compromise*.

Starting from the DMs' individual preferences on the alternatives, NAI classifies the alternatives into three classes of preferences: the most preferable, the preferable and the least preferable. Within each class, relatively small differences in preferences among alternatives may make it reasonable for a DM to consider them interchangeable. This provides the DM with a certain degree of flexibility; the result of this flexibility is that a collective solution acceptable by all DMs can be reached.

After the application of the NAI algorithm, three subsets of alternatives are created from the initial set and each DM's individual ranking: the most preferable, the preferable and the least preferable. It is obvious that, of these three sets, the ones most likely to contain the best compromise are the set of the most preferable and the set of preferable alternatives. Then, on these individual sets an *intersection* operation is performed so that one set of most preferable, one set of preferable and one set of least preferable alternatives

will be created for the *entire group of DMs*. On many occasions, the set of most preferable alternatives contains only one alternative, which allows the decision-making procedure to be completed quite soon. There are, however, decision problems where the set of most preferable alternatives contains more than one alternatives; there are even occasions where one alternative (or more) that appear in some DMs' sets of most preferable alternatives, but not in some others'. In such cases, other techniques are necessary.

It is at this point where the argumentation module comes in; starting from the subset of *preferable* alternatives (instead of the subset of *most preferable*), a multi-round argumentation procedure commences among the multiple agents that represent the DMs. Each agent has its beliefs about the environment and the other agents: each agent has its own ordinal ranking and cardinal preferences (expressed by the utilities) of the alternatives, as well as its own individual weights of the criteria. These are all provided by the UTASTAR method. Then, these agents, which are supposed to be able to make decisions, revise their beliefs and support their points of view using arguments, which are expressed mathematically using the aforementioned utilities and criteria weights, engage in an argumentation-based negotiation using a general protocol, in order to reach a commonly acceptable solution.

Understandably, it can be argued that argumentation as a method is not necessary in multi-criteria decision making and/or multi-criteria decision support; the aggregation functions that can be mimicked in an argumentation-based approach would be considerably simpler than sophisticated aggregation functions (such as a general Choquet integral). There are, however, several reasons for combining MCDA methods and argumentation:

The first reason is the fact that argumentation provides the DMs (or the agents) with the ability to (a) *justify their positions*, (b) *change their positions*.

The second reason is that in some multi-criteria decision problems there are criteria that are intrinsically qualitative in nature; and there are also examples of multi-criteria decision problems where even quantitative criteria (*i.e.* they are of a numerical nature), are perceived in a qualitative manner. An example of this is a decision problem where a DM wants to buy a beach house – in such a problem, the criterion of proximity to the sea, which is quantitative, is often modelled as a qualitative one (Amgoud et al. 2006)

A third reason is the usefulness of developing models that work in a way similar to how humans deal with decision problems. Such models offer DMs tools whose logic they can more easily understand and, therefore, they can more easily accept the end results. Furthermore, it must be noted that argumentation offers a unified setting that can handle inference, as well as decision making under uncertainty. Finally, the logical setting of argumentation offers DMs the opportunity to have the values of consequences of the various alternatives assessed using a non-trivial inference process from various pieces of knowledge, possibly under uncertainty or even partly inconsistent.

I.4 Definitions – Theoretical background

First, a few necessary definitions should be provided, so that the reader will be given a better idea of this work's subject. While there is still no universally accepted definition of an agent, the following definition appears to work well for this thesis' purpose:

- An *agent* is a computer system that is *situated* in some *environment*, and that is capable of *autonomous action* in this environment in order to meet its design objectives (Wooldridge and Jennings 1995).

The present thesis, however, is not concerned with *agents* in general, but with *intelligent agents*. It is not an easy task to provide an answer to questions like *'when do we consider an agent to be intelligent?'* and *'what is intelligence?'*. One way of answering such a question would be to list the kinds of capabilities an intelligent agent would be expected to have. Wooldridge and Jennings (1995) have provided the following list:

- **Reactivity.** Intelligent agents are able to perceive their environment, and respond in a timely fashion to changes that occur in it in order to satisfy their design objectives.
- **Proactiveness.** Intelligent agents are able to exhibit goal-directed behaviour by *taking the initiative* in order to satisfy their design objectives.
- **Social ability.** Intelligent agents are capable of interacting with other agents (and possibly humans) in order to satisfy their design objectives.

Furthermore, this thesis deals with multiple agents as used in the context of a *group decision support system* (GDSS) or *group support system* (GSS); group decision support systems are a subset of the software systems known as *decision support systems*. It is rather hard to give a definition of a *decision support system*. Because of this difficulty, several definitions exist. For instance, Finlay et al. (1994) define a DSS broadly as a computer-based system that aids the process of decision making. Turban (1995) was more precise by defining it as *"an interactive, flexible, and adaptable computer-based information system, especially developed for supporting the solution of a non-structured management problem for improved decision making It utilizes data, provides an easy-to-use interface, and allows for the decision maker's own insights."* Between these two extremes, there are also other definitions. For Keen and Scott Morton (1978), DSSs couple the intellectual resources of individuals with the capabilities of the computer to improve the quality of decisions (*"DSS are computer-based support for management decision makers who are dealing with semi-structured problems"*). For Sprague and Carlson (1982), DSSs are *"interactive computer-based systems that help decision makers utilize data and models to solve unstructured problems."* So, it becomes apparent that a single, universally accepted definition for decision support systems does not exist. Of course, a *group decision support system* (GDSS or GSS) is a DSS designed to support a group of decision makers.

The process of reaching agreements in a multi-agent environment has long been based on *negotiation*. Although it would be fairly easy for someone to confuse *negotiation* with *auction*, these two forms of interaction are quite different. Auctions are indeed useful for allocating goods to agents, but they are too simple for many settings, as they are *only* concerned with the allocation of goods. For reaching agreements on matters of mutual interest, richer techniques for reaching agreements are required. The generic term given to these techniques is *negotiation*. Rosenschein and Zlotkin (1994) have proposed a number of negotiation techniques for use by artificial agents. Before discussing these techniques,

however, it would be useful to say a few things about negotiation in general. Generally speaking, any negotiation setting will have four different components (Wooldridge (2002):

- *A negotiation set*. This represents the space of possible proposals that agents can make.
- *A protocol*. This defines the legal proposals that agents can make, as a function of prior negotiation history.
- *A collection of strategies*: one for each agent – these determine what proposals the agents will make. Usually, the strategy an agent plays is *private*, i.e. the fact that an agent is using a particular strategy is not generally visible to other participants in the negotiation (although most negotiation settings are 'open cry', in the sense that the actual proposals made *are* seen by all participants).
- *A rule*. This determines when a deal has been struck and what this agreement deal is.

Usually, negotiation proceeds in a series of rounds, during which every agent makes a proposal at every round; the proposals made by the agents are defined by their strategy, must be drawn from the negotiation set and must be legal, according to the protocol. If an agreement is reached – as defined by the agreement rule – , then the negotiation process is terminated with the agreement deal.

There are some attributes that determine the complexity of the negotiation. The first is whether *multiple issues* are involved. The most obvious example of a single-issue negotiation is two agents negotiating only on the price of a good that is up for sale. In a scenario like this, the agents' preferences are symmetric, meaning that a deal that is more preferred by one agent is certain to be less preferred by the other – and vice versa. Such scenarios are simple to analyze, as what represents a concession will always be obvious: if the seller is to concede, he must lower the price of his proposal, and if the buyer is to concede, he must raise his proposal's price. Obviously, in *multiple-issue* scenarios, the agents negotiate over the values of multiple attributes, which may or may not be interrelated. In such negotiations, what constitutes a concession is usually much less obvious. There is, however, a considerable drawback: involving multiple attributes in the negotiation makes the space of possible solutions grow exponentially. Also, the complexity of most negotiation domains and the kind of values the attributes in question might have makes things even worse.

Another source of negotiation complexity is the number of agents involved in the process and the way in which they interact. There are three possibilities (Wooldridge 2002):

- **One-to-one negotiation.** Here, one agent negotiates with only one other agent.
- **Many-to-one negotiation.** A single agent negotiates with a number of other agents. An example of this kind of setting is an auction. For analysis' sake,

many-to-one negotiations are often treated as a number of concurrent one-to-one negotiations.

- **Many-to-many negotiation.** Here, many agents negotiate with many other agents simultaneously. In the worst case, with n agents involved in the process in total, there can be up to $n(n-1)/2$ negotiation threads, making the analysis of such negotiations a rather formidable task.

For the above reasons, most attempts to automate the negotiation process have examined simple settings, i.e. single-issue, symmetric, one-to-one negotiations. That is the type of negotiation most commonly analysed in most of the research work, with the possibility of multiple-issue negotiation implied, but usually left unexplored.

Rosenschein and Zlotkin (1994) have described a number of negotiation domains. The first type of negotiation is the *task-oriented domain* (TOD) (pp. 29-52). A task-oriented domain is a triple

$$<T, Ag, c>,$$

where

- T is the (finite) set of all possible tasks;
- $Ag = \{1, \ldots, n\}$ is the (finite) set of agents participating in the negotiation;
- $c = \wp(T) \rightarrow \mathbb{R}^+$ is a function defining the *cost* of executing each subset of tasks: the cost of executing any set of tasks is a positive real number.

The cost function must satisfy the following two constraints: first, it must be *monotonic*. Intuitively, this means that adding tasks never decreases the cost. Formally, this constraint is defined as follows:

If $T_1, T_2 \subseteq T$ are sets of tasks such that $T_1 \subseteq T_2$, then $c(T_1) \leq c(T_2)$.

The second constraint dictates that the cost of doing nothing is zero, i.e. $c(\varnothing) = 0$.

In a task-oriented domain $<T, Ag, c>$, an *encounter* occurs when the agents Ag are assigned tasks to perform from the set T. It can be said, intuitively, that, when an encounter occurs, there is potential for the agents to reach a deal by reallocating the tasks among themselves. Formally speaking, an encounter in a TOD $<T, Ag, c>$ is a collection of tasks

$$< T_1, \ldots, T_n>$$

where, for all i, we have that $i \in Ag$ and $T_i \subseteq T$. A TOD together with an encounter within the TOD is a type of *task environment*, which defines the characteristics of the environment

in which the agent must operate, together with a task – or set of tasks – that the agent must carry out in the environment.

Another type of negotiation domain is the *worth-oriented domain* (WOD). Whereas in a TOD the task(s) are explicitly defined and each agent is given a set of tasks to accomplish, the WOD is more general. The goals are specified through the definition of a *worth* function for the possible states of the environment, which implicitly means that the agent's goal is to bring about the state of the environment with the greatest value. A question that readily occurs is how an agent can bring about a goal. It is assumed that the agents have a set of *joint plans* at their disposal; these plans are joint because the execution of one of them can require the involvement of several different agents. Also, these plans transform one state of the environment to another. To reach an agreement in a TOD, the agents negotiate over a distribution of tasks to agents; here, the agents negotiate over the collection of joint plans and, as mentioned previously, it is in an agent's interest to reach an agreement on the plan that will bring about the state of environment with the greatest worth [Wooldridge, (2002)].

Formally speaking now, a worth-oriented domain (WOD) is a tuple [Rosenschein and Zlotkin, (1994, p. 55)]

$$<E, Ag, J, c>$$

where

- E is the set of possible environment states;
- $Ag = \{1, \ldots, n\}$ is the set of possible agents;
- J is the set of possible environment states;
- $c : J \times Ag \rightarrow \mathbb{R}$ is a cost function, assigning to every plan $j \in J$ and every agent $i \in Ag$ a real number which represents the cost $c(j,i)$ to i of executing plan j.

An *encounter* in a WOD $<E, Ag, J, c>$ is a tuple

$$<e, W>$$

where

- $e \in E$ is the *initial* state of the environment;

$W : E \times Ag \rightarrow \mathbb{R}$ is a *worth* function, assigning to each environment state $e \in E$ and each agent $i \in Ag$ a real number $W(e, i)$ representing the value or worth of state e to agent i.

I.4.1 Advantages of argumentation vs. negotiation

Typically, a decision-making progress utilizing multiple agents involves the implementation of a game-theoretic negotiation system. However, there are several disadvantages to this approach (Jennings et al. 2001):

> **Positions cannot be justified.** When a group of humans negotiates, they *justify* their claims and stances. If someone attempts to sell somebody else a product, the seller will try to justify the price asked, invoking the features it has. In turn, the prospective buyer could justify his proposal for a lower price by explaining that some of the features are not quite necessary to him/her. In general, negotiating through a particular game-theoretic approach may make it difficult to understand *how* an agreement was reached, an issue that becomes all the more important in the e-commerce world, where the task of buying and selling goods is delegated to agents. It makes sense for the owner of the agent to ask *why* the agent paid *this much* for *this* product; if it cannot explain how it reached this agreement in terms its owner can understand and relate to, this agreement will not be easily acceptable. If agents are to act on humans' behalf in this capacity, the human users need to be able to trust their decisions.
>
> **Positions cannot be changed.** In game theory, it is assumed that an agent's utility function is fixed and cannot be changed as the negotiation goes on. This could be true, in one sense, from the point of view of an objective, external, omniscient observer. But not from a real-life human's point of view, which is *subjective* and *personal*. The fact that, in real life, a person has only incomplete information on his/her hands should also be taken into account. So, as the persons negotiate, their preferences *do* change.

It is these limitations of game-theoretic negotiation that have necessitated the emergence of *argument-based negotiation* (Sycara, 1989b, Parsons et al. 1998). In simple words, multi-agent argumentation is a process by which one agent tries to convince another that a state of affairs is true or false (Wooldridge 2002). During the process, agents put forward arguments for and against propositions, combined with justifications for their arguments' acceptability – a process that bears a notable resemblance to the way humans interact to persuade each other of their positions' validity and acceptability.

The philosopher Michael Gilbert suggests that, if argumentation is to be viewed as it happens between humans, at least four different modes of argument (Gilbert 1994) can be identified:

1. **Logical mode.** This resembles mathematical proof. Its nature tends to be deductive ('if we accept that *A* and that *A* implies *B*, then we must accept that *B*). It is perhaps the paradigm example of argumentation and is the kind of argument one expects to see in a court of law or in a scientific paper.
2. **Emotional mode.** This type of argument involves appeals to feelings, emotions and attitudes.
3. **Visceral mode.** This is the physical, social aspect of human argument.

4. **Kisceral mode.** The kisceral mode involves appeals to the intuitive, mystical or religious.

The logical mode is regarded as the 'purest' or 'most rational' kind of argument. Wooldridge (2002) proposed an argumentation system based on the one proposed by Fox et al. (1992) and Krause et al. (1995). It constructs a series of logical steps (arguments) for and against propositions of interest. It closely mirrors the way human dialectic argumentation Jowett (1875) proceeds and so it forms a promising basis for a multi-agent dialectic argumentation framework (Parsons and Jennings 1996).

In classical logic, an argument is a series of inferences that lead to a conclusion. By writing $\Delta \vdash \varphi$ we mean that a sequence of inferences from premises Δ exists that allows us to establish proposition φ. In the argumentation system proposed by Wooldridge (2002), the traditional form of reasoning is extended by explicitly recording the propositions used in the derivation, making the assessment of a given argument's strength possible, by examining the propositions on which it is based. Below, the basic form of arguments is given:

$$Database \vdash \langle Sentence, Grounds \rangle$$

where

- *Database* is a set of logical formulae (possibly inconsistent);
- *Sentence* is a logical formula known as the *conclusion*;
- *Grounds* is a set of logical formulae such as
  1) $Grounds \subseteq Database$; and
  2) *Sentence* can be proven from *Grounds*

Intuitively, it can be said that *Database* is a set of formulae 'agreed upon' among the agents taking part in the negotiation process. This database provides a common ground among the agents. With this common ground given, an agent makes the argument <*Sentence, Grounds*> to support his claim that *Sentence* is true. *Grounds* is a set of formulae such that *Sentence* can be proven from it; thus, *Grounds* provides the justification for the agent's claim that *Sentence* is true.

Formally, if Δ is a database, then an argument over Δ is a pair $\langle \varphi, \Gamma \rangle$ where φ is a formula referred to as the conclusion and $\Gamma \subseteq \Delta$ is a subset of Δ known as the *grounds* or *support*, such that $\Gamma \vdash \varphi$. The set of all such arguments over database Δ is denoted by $A(\Delta)$. *Arg, Arg', Arg*$_1$ … stand for members of $A(\Delta)$.

For a given proposition, an agent can build several arguments; some will be in favour of the proposition, some against it – in the latter case they are for its negation. It is desirable to provide a way to *flatten* the set of arguments into a measure of how favoured

the proposition is, in order to determine whether the set of arguments as a whole are in favour of the proposition. Attaching a numerical or symbolic weight to arguments and then using a flattening function to suitably combine them is one way to do this. Another way is to determine how good the arguments are by using their own structure.

Two important classes of arguments can be identified:

Non-trivial argument. An argument $<\varphi, \Gamma>$ is non-trivial if Γ is consistent.

Tautological argument. An argument $<\varphi, \Gamma>$ is tautological if $\Gamma = \varnothing$.

The idea of defeat between arguments is defined as:

Defeat. Let $<\varphi_1, \Gamma_1>$ and $<\varphi_2, \Gamma_2>$ be arguments from some database Δ. The argument $<\varphi_2, \Gamma_2>$ can be defeated in one or two ways. First, $<\varphi_1, \Gamma_1>$ *rebuts* $<\varphi_2, \Gamma_2>$ if φ_1 attacks φ_2. Second, $<\varphi_1, \Gamma_1>$ *undercuts* $<\varphi_2, \Gamma_2>$ if φ_1 attacks ψ for some $\psi \in \Gamma_2$ (Wooldridge, 2002).

Defining attack:

Attack. For any two propositions φ and ψ, φ attacks ψ if and only if $\varphi \equiv \neg \psi$ (Wooldridge 2002).

Walton and Krabbe (1995) suggested a typology of six different modes of dialogues, which are summarized in Table 1. The first mode (type I) is the 'canonical' form of argumentation; i.e. an agent tries to convince another that something is true, while deliberation (type IV) seems to be the mode closest to the needs of the decision-making process.

Type	Initial situation	Main goal	Participant's aim
I. Persuasion	conflict of opinions	resolve the issue	persuade the other
II. Negotiation	conflict of interests	make a deal	get the best for oneself
III. Inquiry	general ignorance	growth of knowledge	find a 'proof'
IV. Deliberation	need for action	reach a decision	influence outcome
V. Information seeking	personal ignorance	spread knowledge	gain or pass on personal knowledge
VI. Eristics	conflict/antagonism	reaching an accommodation	strike the other party
VII. Mixed	various	various	various

Table 1: The dialogue types as defined by Walton and Krabbe [Walton and Krabbe (1995)]

Chapter II *THE PROPOSED METHODOLOGY*

This chapter presents, and elaborates on, the methodology proposed for the solution of the decision problem at hand. More specifically, the structure and logic of the methodology is presented and documented. This chapter is structured as follows: i) it presents the general methodological framework; this presentation is divided in the Ranking Stage, the NAI algorithm and the Argumentation stage. ii) All three stages are divided and organized in steps, which are presented and described.

II.1 General methodological framework

A group faces the problem of selecting an action from a set of actions (alternatives). These alternatives presented to the group are valued by a family of criteria. Let $A = \{\alpha_1, \alpha_2, \ldots \alpha_n\}$ be the set of alternatives, $g = \{g_1, g_2, \ldots g_m\}$ the consistent family of criteria and $D = \{d_1, d_2, \ldots d_q\}$ the decision makers that form the group. The choice problem is divided into three sub-problems, which are solved consecutively, with the first sub-problem's output being the second sub-problem's input and the second sub-problem's output being, eventually, the third problem's input.

In the first sub-problem, the aim is to form each individual DM's ordinal rank order of the alternatives according to the problem's given criteria and the views of each DM. In the second sub-problem, a possible set of negotiable alternatives is produced from the different ordinal rankings of the DMs using the Negotiable Alternatives Identifier (NAI) algorithm. Finally, in the third sub-problem, a multi-criteria argumentation framework is applied on the set of negotiable alternatives so that an alternative will be chosen. These three sub-problems will be called *stages* for the rest of this thesis. Thus, the decision problem becomes a three-stage problem. These stages are:

- The *Ranking stage*
- *The Negotiable Alternatives Identification stage*
- The *Argumentation stage*

II.1.1 The Ranking stage

In this stage, the proposed system solves the problem of ranking the alternatives for each individual DM. This stage consists of the following phases:

II.1.1.1 Setup phase

In this phase, the alternatives and criteria are defined and the group norms (i.e. the procedures by which the group of DMs will operate) are determined.

II.1.1.2 Assessment of the preferences of the group members

A rank order of the alternatives is constructed according to the preferences of the individual DMs through the application of the UTASTAR method (Siskos and Yannacopoulos 1985). The UTASTAR method has been chosen primarily because it enables each individual DM to analyze his/her behavior and cognitive style according to the general preference disaggregation framework (Jacquet-Lagrèze and Siskos, 1982, Siskos 1980, 1985, Siskos and Yannacopoulos 1985, Siskos et al. 1993). This approach aims to help the DM improve his/her knowledge about the decision situation and his/her preference/value system in order to reach a consistent decision. In the UTASTAR method, it is assumed that the model of the DM's preferences is additive, which is not always true, as there are decision problems where this assumption does not apply. However, the assumption of a linear preference system helps in the simplification of the problem and also makes it easier to assess the DM's preference system.

II.1.1.3 Calculation of relative utility values for each alternative and DM

In this phase, the utilities of each alternative (which are derived in phase 2) are normalized, so that a relative utility value that reflects the tendency of each DM to select or reject an alternative will be calculated.

II.1.1.4 Ranking of the alternatives

In this phase, a disaggregation of the relative utility values is performed and, taking into consideration any special knowledge or expertise of each DM, an individual rank order of the alternatives is constructed for each DM.

II.1.2 Elaboration on the aforementioned phases

II.1.2.1 Setup phase

In this phase, the DMs that form the group are called upon to decide on an initial set of alternatives and criteria, which may be altered during the process. The already existing

literature (Keeney and Raiffa 1976, Keeney 1992, Roy 1996, Kirkwood 1997) provides a thorough discussion of guidelines, methods and tools for the selection of a set of alternatives and the construction of a consistent family of criteria. The common set of dimensions (alternatives and criteria) that was used in the case of MEDIATOR (Jarke et al. 1987) can be used to provide a representation of the group decision problem.

In real life, it is not uncommon to accept the fact that, for certain tasks, certain members of the decision-making group are more qualified for the selection of the final decision because of factors such as knowledge, expertise, skills etc. This attitude is adopted in the proposed methodology and it is modeled by granting each DM with a decision power b_k, which represents each participant's ability to influence the outcome of the decision-making process.

The decision power variables offer the advantage of respecting the particular characteristics and abilities of each DM, while ensuring that the decision will be made collectively through the participation and cooperation of all the group members. In practical collective decision environments, this type of weighted scheme appears frequently (Laruelle and Widgren 2000, Van Houtven 2002 Turnovec 2002, Leech 2002, Uno 2003, Felsenthal and Machover 2004) and it has been included in decision support systems Csàki et al. (1995a, 1995b). Barzilai and Lootsma (1997) point out that, although many decisions are made in boards, committees and councils and not by individual DMs, little attention is paid by multicriteria decision analysis to the power relations in groups, although power games are always present. According to the authors, if the assumption is made that all group members share equal weights then alternatives weakly supported by the "powerful" members have little chance of being eventually adopted by the group, even if they are well-supported by a multicriteria analysis. However, in the case of this thesis, the DMs are considered to have equal decision powers for the sake of simplicity. Furthermore, the DMs are considered to be working in a cooperative context.

II.1.2.2 Assessing the preferences of the group members

Following the determination and the formulation of the problem (*i.e.* alternatives, criteria and decision powers), the process continues with the assessment of each individual DM's preferences. Although all individual DMs should share the same set of criteria, each DM can assign different weights to each criterion. So, if a DM considers a certain criterion to be unimportant, s/he can assign zero weight to it. The weights of the criteria are expressed implicitly in each DM's assessment and are calculated by the UTASTAR method.

For this stage of the methodology, a ranking problematic has been chosen for the following two reasons:

1. According to Roy (1985), there are four reference problem statements, each of which does not necessarily preclude the others:
   (a) Choosing one action from A (choice) – problematic a
   (b) Sorting the actions into predefined and preference-ordered categories (sorting) – problematic β
   (c) Ranking the actions from the best to the worst (ranking) – problematic γ
   (d) Describing the actions in terms of their performances on the criteria

(description) – problematic δ

2. Group decision making offers the advantage of the potential to gather and combine the information to which each group member has access (Dose 2003).

In a study of rank-order effects (Hollingshead 1996), groups have been found to be more likely to exchange information and to consider all alternatives thoroughly when members were asked to rank order alternatives rather than simply choosing the best one. Thus, the ranking problematic appears to satisfy the needs of group decision making quite well.

The UTASTAR multicriteria method (Siskos and Yannakopoulos 1985) is applied on the preferences expressed on the set of alternatives in order to capture each group member's preferences. With a weak-order preference structure ($\succ$, ~), where $\succ$ signifies the strict preference and ~ the indifference on a set of actions or objects, the method aims to adjust additive utility functions based on multiple criteria in such a way that the resulting preference structure will be as consistent as possible with the initial structure. In the original context in which UTA and its later variations (one of which is the UTASTAR method) were developed, the additive utility model is estimated based on an ordering of a reference set; the results are extrapolated to the complete set of alternatives. In the problem at hand, though, the DMs are asked to provide their evaluation on the complete set of alternatives. This approach will not cause any problems in most situations, except in cases where there is a large number of alternatives.

Each criterion is defined under the form of a real-valued monotone function $g_i : A \rightarrow [g_{i*}, g_i^*] \subset R$ *in such a way that* $g_i(\alpha)$, $\alpha \in$ A represents the evaluation of the action a on the criterion g_i and g_{i*}, g_i^* respectively represent the level of the most desirable and the least desirable criterion.

The UTASTAR regression's aim is to estimate additive utilities:

$$u(g) = u_1(g_1) + u_2(g_2) + \ldots + u_m(g_m) \quad (1)$$

satisfying that

$$u_i(g_{i*}) = 0 \ \forall \ i \quad (2)$$

$$\sum_{i=1}^{m} u_i(g_i^*) = u_1(g_1^*) + u_2(g_2^*) + \ldots + u_m(g_m^*) = 1 \quad (3)$$

The UTASTAR method guides the DM to a process of gradual learning of his preferences. The solution is the one that maximizes the DM's satisfaction. What happens, though, if the results produced by the model are not in agreement with the individual DM? In this case, several types of feedback to Phase 1 (the Setup phase) can be considered. Having returned to the Setup phase, the DM can:

- Change the criteria

- Change the weak order of the alternatives
- Make tradeoffs among the criteria

In order to assess every marginal value function, the evaluation scales of each criterion (especially in the case of quantitative criteria that are easily measurable) are discretised in a limited set of points:

$$G_i = \{g_i^1, g_i^2, \ldots, g_i^l, g_i^{a_j} = g_i^*\} \quad (4)$$

On the other hand, the set of actions (alternatives) $A = \{a_1, a_2, \ldots, a_k\}$ is rearranged in such a way that a_1 is the head of the ranking and a_k is its tail. Since the ranking has the form of a weak order, for each pair of consecutive actions (a_j, a_{j+1}) one of the two following relations holds:

$$a_j \succ a_{j+1} \text{ (preference)}$$

$$a_j \sim a_{j+1} \text{ (indifference)}$$

Whereas the original UTA method introduced a single error $\sigma(a)$ to be minimized for each $a \in A$, the UTASTAR method introduces two errors leading to better results (Fig. 1).

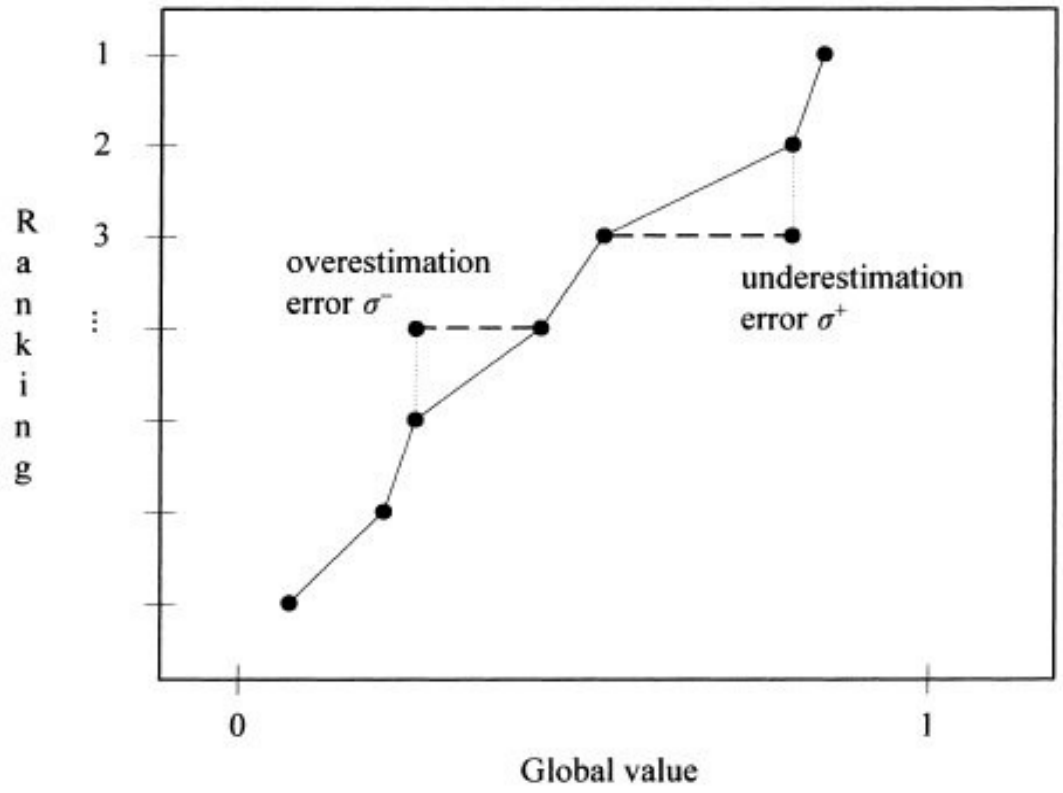


Figure 1. Ordinal regression curve (ranking versus global value)

The main computational procedure employed in UTASTAR uses linear programming techniques to find additive value (utility) functions that are as consistent as possible with the ranking on A.

Step 1. Express the global value (global utility) of the alternatives $u[g(a_j)]$, $j = 1, 2, \ldots, k$, first in terms of marginal values (marginal utilities) $u_i(g_i)$ and then in terms of variables:

$$w_{il} = u_i(g_i^{j+1}) - u_i(g_i^l) \geq 0 \quad , \tag{5}$$

$$i = 1, 2, \ldots, n,\ l = 1, 2, \ldots, a_i - 1, \tag{6}$$

by means of the relations

$$u_i(g_i^1) = 0 \quad \text{and} \quad u_i(g_i^l) = \sum_{i=1}^{l-1} w_{il} \quad \forall\ i \text{ and } l > 1. \tag{7}$$

Step 2. Introduce two error functions, σ^+ and σ^- on A by writing, for each pair of consecutive alternatives in the ranking, the analytic expressions

$$\Delta(a_j, a_{j+1}) = u[g(a_j)] - \sigma^+(a_j) + \sigma^-(a_j) - u[g(a_{j+1})] + \sigma^+(a_{j+1}) - \sigma^-(a_{j+1}) \tag{8}$$

Step 3. Solve the linear program

$$\text{minimize } z = \sum_{j=1}^{k} [\sigma^+(a_j) + \sigma^-(a_j)] \quad , \tag{9}$$

subject to the set of constraints:

$$\Delta(a_j, a_{j+1}) \geq \delta \quad \text{if } a_j \succ a_{j+1}$$

$$\Delta(a_j, a_{j+1}) = 0 \quad \text{if } a_j \sim a_{j+1}$$

$$\forall j = 1, 2, \ldots, k-1 \quad , \tag{10}$$

$$\sum_{i=1}^{n} \sum_{l=1}^{a_i - 1} w_{il} = 1 \quad , \tag{11}$$

$$w_{il} \geq 0, i = 1, 2, \ldots, n \quad , \tag{12}$$

$$l = 1, 2, \ldots, a_i - 1 \quad , \tag{13}$$

$$\sigma^+(a_j) \geq 0, \sigma^-(a_j) \geq 0, j = 1, 2, \ldots, k \quad , \tag{14}$$

where δ is a small positive number.

Step 4. Test the existence of multiple or near optimal solutions of the linear program (9)-(14) (stability analysis). In case of non-uniqueness, find the mean additive value function of those (near) optimal solutions that maximize the objective functions $p_i = u_i(g_i^*) = \sum_l w_{il} \; \forall i = 1, 2, \ldots, n$ on the polyhedron (10)-(14) bounded by the new constraint

$$\sum_{j=1}^{k} [\sigma^+(a_j) + \sigma^-(a_j)] \leqslant z^* + \varepsilon \quad , \tag{15}$$

where z^* is the optimal value of the linear program in Step 3 and ε is a very small positive number. Fig. 2 on the next page illustrates the Ranking Stage in the form of a flowchart.

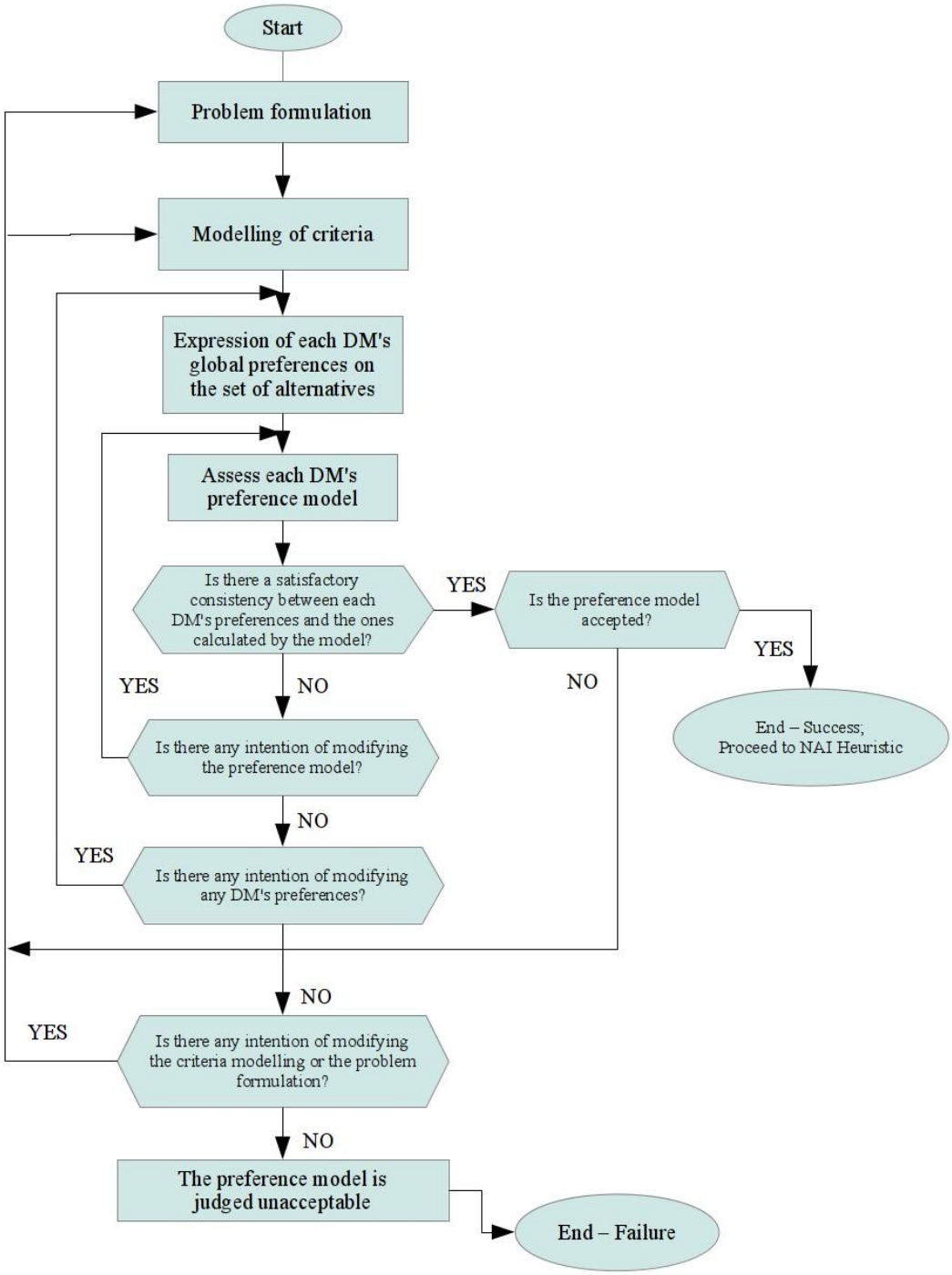


Figure 2. Illustration of the Ranking Stage.

II.1.3 Identification of Negotiable Alternatives

After the assessment of each DM's preferences in the previous stage has been completed, all group members will have reached a final ranking of the alternatives, which is consistent to their initial preferences and the original weak order. There is, however, one problem: since DMs have different preferences, it is not uncommon to encounter conflicts and disagreements among the individual rankings; this prevents the selection of a unique and commonly accepted group ranking of the alternatives.

This necessitates the usage of a method that will seek a point of consensus, i.e. a compromise among the possibly conflicting individual rankings. This is where a heuristic algorithm named the Negotiable Alternative Identifier (NAI) algorithm proves to be especially useful. NAI classifies the alternatives into three classes of preferences: the most preferred, the preferred and the least preferred. Within each class, relatively small differences in preferences (in the case of this thesis' proposed methodology, preferences are expressed by the utility values of the alternatives) among alternatives may make it reasonable for a DM to consider them more or less interchangeable.

As a result of this flexibility, the generation of a collective solution or a set of collective solutions that are acceptable to all DMs becomes possible. The NAI algorithm, which entails multiple rounds of group consideration, is a formalized consensus-seeking methodology based on an intuitive procedure observed in negotiations. This procedure is typically described as follows: "The group members have failed to find a consensus. However, if some are willing to accept solutions other than their first choice, but which are not that far different preference-wise from the first choice, then a common solution that is acceptable by all members can be found". The underlying concepts of the NAI algorithm shall be discussed in section II.1.3.1. In section II.1.3.2 the NAI methodology and its mathematical model shall be described.

II.1.3.1 Consensus-Seeking: Problem definition and basic concepts

II.1.3.1.1 Definition of the Problem

1. All DMs share the same exhaustive and mutually exclusive alternatives, where n is the number of alternatives and m is the number of DMs participating in the solution of the group decision problem.
2. Prior to the group decision making process, each DM d has performed his/her own individual assessment of preferences (the *Ranking Stage* described earlier). For example, the DM can use an additive utility method [Fishburn, (1974a, b), Siskos and Yannakopoulos (1985)] or perhaps the Analytic Hierarchy Process [Saaty, (1980)] to obtain utilities. The output of this analysis is a vector of normalized cardinal preferences on n alternatives $\mathbf{r_d} = [r_{di}]$, where $r_{di} \geq 0$ for $i = 1, \ldots, n$, $d = 1, \ldots, m$, and $\sum_{i=1}^{n} r_{di} = 1$

3. Furthermore, the vector of ranking, $\mathbf{r_d}$, is sorted according to an order of decreasing importance. This notion of preference corresponds to a complete asymmetry preorder. In other words, r_{d1} represents the relative preference of the most preferred alternative and r_{dn} the relative preference of the least preferred alternative.
4. Given the vector $\mathbf{r_d}$, R_{ds} can be defined as the cumulative preference that a decision maker gives to the first *s* alternatives:

$$R_{ds} = \sum_{i=1}^{s} rdi \qquad (16)$$

II.1.3.1.2 Expansion/Contraction/Intersection Concept

Beginning with individual and cardinal rankings of the alternatives, the NAI algorithm is motivated by the following observations: First, the possibility of reaching a consensus can be improved if the DMs exhibit some flexibility regarding their individual assessment of preferences. Second, they should be able to identify exchangeable or negotiable alternatives.

As proposed by Bui and Shakun (1988), the NAI algorithm, which attempts to help DMs who exhibit flexibility in their assessment of the preferences , is based on the observations that the determination of the cardinal ranking of a set of alternatives is influenced by two factors:

1. The total number of the alternatives that are being evaluated affects the intensity of preferences. Often, the greater the number of alternatives, the weaker the relative importance of the alternatives is. This means that, when the DMs are presented with a greater number of alternatives, it becomes more difficult for them to tell which one is more important (or better) than the other.
2. The distribution of marginal difference among the alternatives is rarely uniform. For example, some alternatives share close evaluation (e.g., A and B with respective scores 0.33 and 0.32), while others score significant marginal difference (for instance, C and D with 0.25 and 0.11 respectively).

NAI is characterized by a triplet of operations: expansion, contraction and intersection. The objective of the first operation is to assess individual preferences by locating possible areas of compromise. In effect, when a DM ranks his/her preferences, the order is constantly subject to re-evaluation. He/she logically chooses the alternative that is ranked first; however, he/she may consider others, depending on their relative distances from the first.

NAI groups the alternatives (which, in this thesis and its accompanying software, have already been ranked by the UTASTAR method in the *Ranking Stage*) into three classes of preferences: the most preferable, the more preferable and the least preferable subsets. Within each class, negligible differences in preferences among alternatives would increase the confidence of the DMs not to discriminate among them. Consequently, it would make it easier for the DMs to trade them interchangeably. In other words, grouping alternatives that share close evaluation corresponds to *expanding* the preference space(s) of the DM from one best alternative to a set of more or less equally preferred ones.

The *contraction* operation constitutes the second phase of the NAI algorithm. Given a subset of comparatively satisfactory alternatives obtained from the expansion mapping, the second operation attempts to identify those that might exhibit a stronger preferential distribution than others. Therefore, if among the preferred alternatives, there still remains an unequal distribution of preferences, the NAI algorithm provides an indicator that distinguishes the most preferable alternatives from the preferable ones.

Finally, the third and last step of the NAI algorithm is the *intersection* operation. It derives a collective solution (or a number of collective solutions) that is (are), in principle, acceptable by all group members. Consensus is reached when there is at least one alternative that appears in every group member's subset of the most preferable alternatives. As a result, a collective solution is one that is acceptable by all the DMs to whom it may be suggested.

If, however, the intersection operation fails to identify a collective solution, this could be seen as an indicator that another form of consensus seeking or compromise should be tried.

II.1.3.2 Heuristics for Consensus Seeking

II.1.3.2.1 NAI Heuristic

As mentioned earlier, the distribution of preferences among alternatives reflects the extent to which the alternatives are related to each other. With alternatives ranked by cardinal preferences, the *Structural Index of Preferences* of the Subset consisting of the first *j* alternatives, Si_{dj}, can be defined as follows. For a decision maker *d*,

$$SI_{dj} = \left(\frac{1}{j}\right) M_d^{(j)} \quad , \tag{17}$$

where

$$M_d^{(j)} = \left(\frac{1}{j-1}\right) \sum_{k=1}^{j-1} M_{dkj} \quad , \tag{18}$$

and

$$M_{dkj} = \frac{\left(\frac{R_{dk}}{k}\right)}{\frac{R_{dj} - R_{dk}}{j-k}} \quad , \tag{19}$$

where $j = 2, \ldots, n$ and $k = 1,..., j\text{-}1$ is a summation index. Here, it is assumed that the denominator is not zero. Otherwise, it is assumed that $M_{d\,k\,j} = \infty$.

In other words, $M_{d\,k\,j}$ is the ratio between the cumulative preference per alternative assigned to better alternatives and that of the remaining alternatives. $M_d^{(j)}$ is an average value of M_k. The structural index SI_{dj} puts this average $M_d^{(j)}$ on a per-alternative basis.

The value of SI_{dj} is a function of the number of alternatives j, as well as the distribution of the DM's preferences r_{di}. In theory, SI_{dj} varies between $\frac{1}{j}$ (i.e., a situation in which the DM is completely indifferent w.r.t the alternatives) and ∞ (i.e., the maximum "disequilibrium" or imbalance in the distribution of preferences). Furthermore, it can be argued that the closer SI_{dj}'s value is to $\frac{1}{j}$, the easier it is for the DM to negotiate with other members of the group. On the other hand, the degree of flexibility in negotiation becomes smaller as the value of SI_{dj} becomes higher.

The NAI algorithm's function is broken down in three basic operations:

Operation 1: Expansion. Given a set of n ranked alternatives, the subset of preferable alternatives can be defined as the one consisting of the top alternatives, say n^*, that are clearly more preferable to the others, i.e. $n - n^*$. The identification of the number of preferable alternatives n^*, as well as the reasoning of the approach, are described below:

- Define $n - 1$ subsets of alternatives: the first subset is composed of the first two alternatives ($j = 2$). The second is composed of the first three alternatives ($j = 3$), etc. The $(n - 1)$th subset is the entire set of alternatives itself ($j = n$).
- For each subset of j alternatives, compute its structural index of preferences, SI_{dj}, where $j = 2, \ldots, n$.
- The subset containing the preferred alternatives is the one that has the lowest SI_{dj}:

$$SI_{d,n^*} = min\{SI_{d\,j}\} \tag{20}$$

where n^* represents the first n^* alternatives that form the subset of the preferred alternatives ($2 \le n^* \le n$).

But why should the lowest value of SI_{dj} be chosen as the cut-off point? This choice can be intuitively justified by observing that the lower the SI_{dj}'s value is, the more uniform the distribution of preferences among alternatives becomes. Thus, by choosing n^* that has the minimum value of SI_{dj}, it can be inferred that the DM d has distributed his/her preferences among the n^* alternatives more or less evenly. In other words, numerical differences between the n^* alternatives are not significant enough to assert that none of the alternatives is clearly worse than another to the extent that it should be rejected.

From a group decision problem solving point of view, a higher SI_{d,n^*} value indicates that the DM d has a strong and clear choice. Consequently, there may be little room left for concession. On the other hand, a low SI_{d,n^*} suggests that the DM d would exhibit some

indifference to the alternatives, and therefore any of these could be acceptable.

Operation 2: Contraction. In this operation, the idea is to find out which subset of the preferred set constitutes the most preferred subset. Given n^* preferred alternatives, a second cut-off point can be identified by following the steps set forth below:

- Define n^* - 1 subsets of alternatives in a bottom-up fashion: the first bottom subset is composed by the n^* alternatives minus the top one (the one that is at the top of the ranking). The second subset is composed by n^* alternatives minus the first two top alternatives, and so forth. Finally, the (n^* - 1)th bottom subset contains only one alternative, the one just above the cut-off point for the preferred set.
- Compute the arithmetic mean $r_{\bar{i}}$ of the cardinal preferences of each subset i', where $i' = 1, \ldots, n^* - 1$ corresponds respectively to the first to the (n^* - 1)th bottom subset as defined in the first step.
- For $i^* = 1, \ldots, n^* - 1$, compute the preference ratio index, C_{d,i^*} as follows:

$$C_{d,i^*} = \frac{r_{i^*}}{r_i} \tag{21}$$

  where r_{i^*} is the cardinal preference for alternative i^*, the last top alternative defining a cut-off point separating the bottom subset from the alternatives above. It must be noted that the cardinal preferences of the alternatives in the preferred set, n^*, are renormalized so that their sum equals one.
- Choose the second cut-off point i^* by maximizing the C_{d,i^*} preference ratio, i.e., $\max\{C_{d,i^*}\}$ for $i^* = 1, \ldots, n^* - 1$. The rationale for this is as follows: If C_{d,i^*} is large, then there is a significant relative drop between the preference value r_{i^*} of the alternative just above the cut-off point compared to the average preference r_i of the alternatives in the subset below. Thus, $\max\{C_{d,i^*}\}$ is a good criterion for the subset of the preferred alternatives at the top of the preferred set.

In other words, the alternatives situated above this second cut-off point are considered to be the most preferable. It is assumed that the DM would be reluctant to reject them. In a situation of complete indifference, all $C_{d,i^*} = 1$ are maximum and the algorithm would set $i^* = n^*$.

Operation 3: Intersection. Given all individual subsets of i^* (most preferable) alternatives, an intersection operation can be performed to identify a possible consensus solution (or more possible consensus solutions, if they do exist). In a similar manner, an intersection operation can be performed on individual subsets of n^* (preferable) alternatives.

II.1.3.2.2 The Intersection Impasse and two procedures to overcome it

The above has proved to be useful, since it captured, at least partially, the human behavior in decision making (Bui, 1987). If each DM selects his/her own i^* and deletes all the others,

then there might be no common alternative in every DM's most preferred set i^*. This means the following:

$$i_1^* \cap i_2^* \cdots i_m^* - 1 \cap i_m^* = \{\} \tag{22}$$

where m is the number of DMs. In other words, the most preferred alternatives have an empty intersection. To overcome this difficulty, Bui and Yen (1995) propose two procedures:

Procedure 1. This procedure is an iterative process. In every iteration, it removes the alternatives that are the least likely to be accepted as candidates, i.e. if no common alternative is identified in every DM's most preferred set i^*, then the alternatives that fall in the least preferred set, the $n - n^*$, will be removed and the preferences distribution will be reallocated to the ines remaining in the more preferred set n^*.

In every iteration, the procedure recalculates r_i, SI_{dj} and $C_{d,i*}$ to find the new cut-off points to divide the n into n^* and n - n^* and to divide the n^* into i^* and n^* - i^*. This procedure continues until a common alternative (or a number of common alternatives) is identified in every DM's most preferred set i^*. The procedure assumes that the preference distribution is indifferent to the removal of the least preferred alternatives. There is, however, no general guarantee that this procedure will lead to a consensus.

Procedure 2. This procedure differs from procedure 1 in that it expands the size of the most preferred set downward until an alternative is identified in every DM's most preferred set. The length of the procedure depends on the distribution of preferences. If the allocations do not deviate widely, reaching a solution only takes two steps. Otherwise, completion of all four steps presented below is required.

Step 1. For each DM d, find the following two bounds:

$$C_{d,min} = min_{i \in (n^* - i^*)} \{C_{di}\} \tag{23}$$

$$C_{d,max} = max_{i \in (n^* - i^*)} \{C_{di}\} \tag{24}$$

These are the upper and lower limites of C_{d,n^*-i^*} .

Step 2. For each DM d and value $t \in [0,1]$, identify $i_d(t)$, which is the largest value of i such that

$$C_{d,i} \geq C_{d,max} - t \cdot (C_{d,max} - C_{d,min}) \tag{25}$$

These steps represent the fact that because the DMs cannot find an alternative that appears in everyone's most preferred set i_d^* , they have to further expand the size of the most preferred set by including part of the alternatives in the more preferred set $n^* - i^*$. Then, the lowest value of t is selected such that the DMs have at least one common alternative of all expanded most preferred sets, and any one of the common alternatives can be accepted as the solution.

Application of the above two steps will always provide a solution unless there is no common alternative in the preferred sets of the DMs. In such a case, the preferred sets should be expanded again before applying the above procedure. The expansion can be done as follows:

Step 3. For each DM d, find the following two bounds:

$$SI_{d,min} = min_{j \in (n-n^*)} \{SI_{dj}\} \quad (26)$$

$$SI_{d,max} = max_{j \in (n-n^*)} \{SI_{dj}\} \quad (27)$$

Step 4. For each DM d and value $t \in [0,1]$, calculate $j_d(t)$, which is the largest value of j, such that

$$SI_{d,j} \leq SI_{d,min} + t \cdot (SI_{d,max} - SI_{d,min}) \quad (28)$$

The smallest value of t is selected such that the expanded preferred alternative sets have at least one common element. Steps 1 and 2 are then used with the expanded preferred alternative sets.

The aforementioned procedures were Bui and Yen's propositions for breaking a possible intersection impasse. Although this thesis uses the NAI algorithm as a method for *accelerating* and *verifying* the argumentation process and not as the main method of reaching a consensus, both these procedures are implemented in order to ensure that no empty intersection occurs; that way, it is guaranteed that the argumentation stage will have alternatives on which the agents will negotiate.

On the next page, figure 3 presents the NAI algorithm in the form of a flowchart.

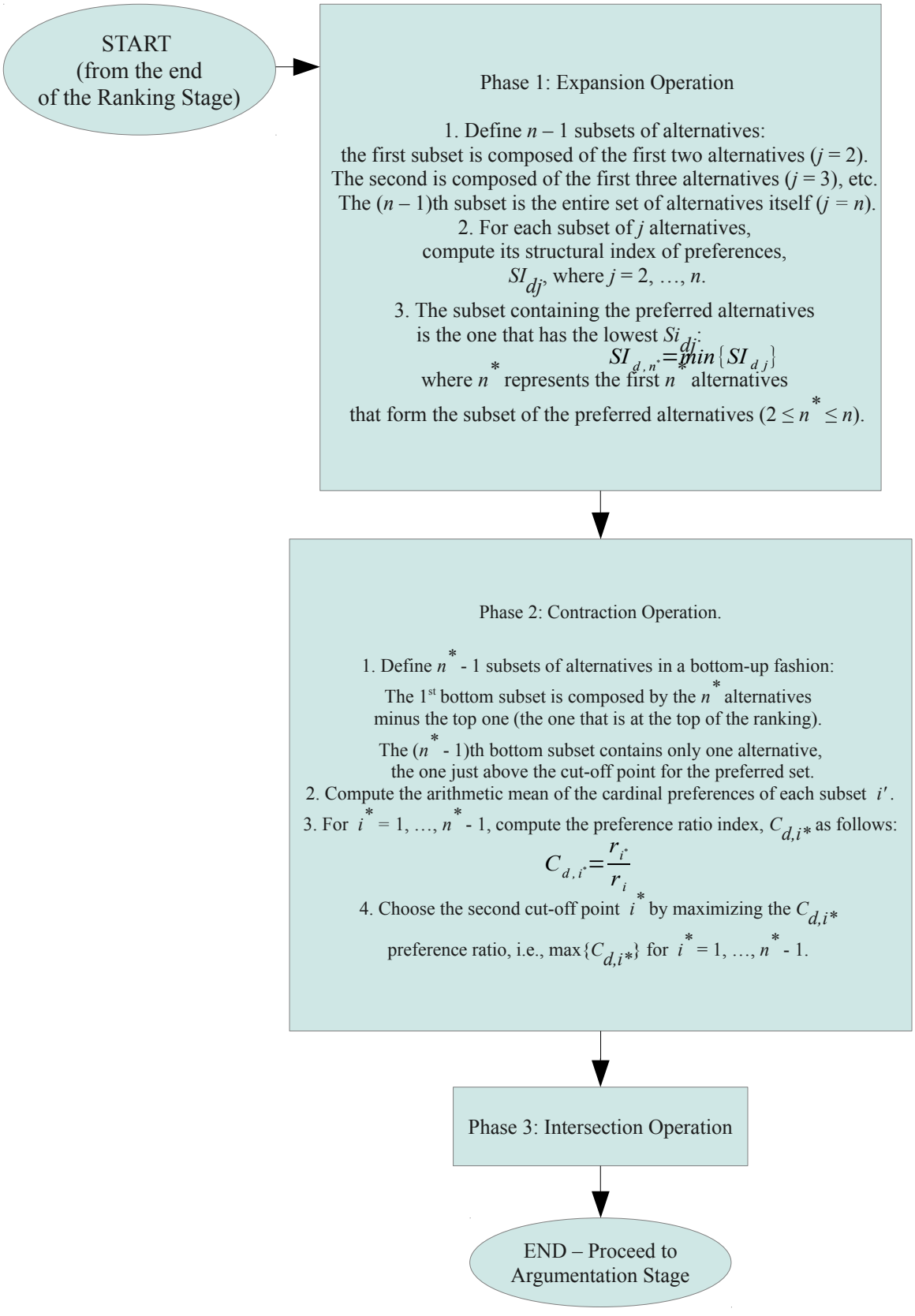

START
(from the end
of the Ranking Stage)
Phase 1: Expansion Operation
1. Define $n - 1$ subsets of alternatives:
the first subset is composed of the first two alternatives ($j = 2$).
The second is composed of the first three alternatives ($j = 3$), etc.
The $(n - 1)$th subset is the entire set of alternatives itself ($j = n$).
2. For each subset of j alternatives,
compute its structural index of preferences,
SI_{dj}, where $j = 2, \ldots, n$.
3. The subset containing the preferred alternatives
is the one that has the lowest Si_{dj}:
$SI_{d,n^*} = \min\{SI_{d,j}\}$
where n^* represents the first n^* alternatives
that form the subset of the preferred alternatives ($2 \le n^* \le n$).
Phase 2: Contraction Operation.
1. Define $n^* - 1$ subsets of alternatives in a bottom-up fashion:
The 1st bottom subset is composed by the n^* alternatives
minus the top one (the one that is at the top of the ranking).
The $(n^* - 1)$th bottom subset contains only one alternative,
the one just above the cut-off point for the preferred set.
2. Compute the arithmetic mean of the cardinal preferences of each subset i'.
3. For $i^* = 1, \ldots, n^* - 1$, compute the preference ratio index, C_{d,i^*} as follows:
$C_{d,i^*} = \frac{r_{i^*}}{r_i}$
4. Choose the second cut-off point i^* by maximizing the C_{d,i^*}
preference ratio, i.e., $\max\{C_{d,i^*}\}$ for $i^* = 1, \ldots, n^* - 1$.
Phase 3: Intersection Operation
END – Proceed to
Argumentation Stage

Figure 3. The NAI Algorithm

II.1.4 The Argumentation Stage

As has been mentioned earlier, the NAI algorithm is used as a method for *accelerating* the reaching of a consensus on a commonly accepted solution through an argumentation-based negotiation procedure. What the NAI algorithm basically does in this context is to *reduce* the number of possible alternatives from the initial set of n alternatives to a smaller set of preferable (i.e. *negotiable*) alternatives n^*. This provides an opportunity to complete the argumentation process in fewer iterations (rounds), having excluded all the alternatives that the DMs do not consider negotiable.

It is in this stage that the collective choice is being made by the DMs (and the agents that represent the DMs). To apply this approach, this thesis adapts an argumentation framework proposed by Amgoud, Belabbes and Prade (2005), using the output of the NAI algorithm as its input. The setting of the framework used in this thesis has three main components: the agents/DMs, their reasoning capabilities and a protocol. The agents/DMs are supposed to maintain certain beliefs about their environment and other agents/DMs, together with their own goals. These beliefs are generally more or less certain and the goals may or may not have equal priority. Furthermore, the agents/DMs are supposed to be able to make decisions, revise their beliefs and support their point of view using arguments. The protocol used in this thesis governs the high-level behavior of the interacting agents/DMs, specifying the legal moves in the dialogue.

Argumentation is especially useful, as it helps explain and support the choice made by the group of decision makers, not only providing the users with a "good" choice, but also with the reasons underlying the recommendation made by the GDSS, in a format easily comprehensible by the average decision maker, who could very well be a layman, without significant knowledge on matters of Operations Research, Game Theory, Argumentation and Artificial Intelligence. Furthermore, argumentation-based decision making is similar to the way humans deliberate and finally make their decisions. Of course, the idea of basing decisions on arguments pro (*for* the alternative) and cons (*against* the alternative) is not new at all; it is very old and was actually stated in a more or less formal fashion by Benjamin Franklin more than two centuries ago.

Several attempts have been made to formalize this idea; the most important ones were by Fox and Parsons (1997), Fox and Das (2000), Bonet and Geffner (1996), Amgoud and Prade (2004), Amgoud, Belabbes and Prade (2005) and Amgoud, Bonnefon and Prade (2005).

The proposed approach applies the ideas presented by Amgoud, Belabbes and Prade (2005) and incorporates them as the *argumentation stage* of the presented methodology and software. The argumentation stage presented here is basically an automated system for argumentation-based negotiation. Automated negotiation is not a new concept; Rahwan et al. (2003) have investigated the matter before and various approaches have been proposed, including *game-theoretic* approaches (these usually assume complete information and unlimited computation capabilities), *heuristic-type* approaches that attempt to cope with these limitations, and *argumentation-based* approaches, such as those presented in Amgoud et al. (2000a, 2000b), Kakas and Moraïtis (2003), Kraus, Sycara and Evenchik (1998) and

Parsons, Sierra and Jennings (1998), which emphasize the importance of exchanging information and explanations between negotiating agents in order to mutually influence their behaviors (e.g. an agent may concede a goal having a small priority). The first two types of settings do not allow for the addition of information or for exchanging opinions about offers. Integrating argumentation theory in negotiation provides a good means of supplying additional information and also helps agents convince each other by using adequate arguments during a negotiation dialogue.

The argumentation stage presented here considers agents/DMs having knowledge about the environment graded in certainty levels and preferences expressed under the form of more or less important goals. These goals can intuitively be identified with the criteria of the GDSS. Thus, the criteria provided in the early stage (the *Ranking Stage*) of this GDSS are the goals used in the *Argumentation Stage* presented and discussed here. The reasoning model of the agents/DMs is based on an argumentative decision framework, as the one proposed in Amgoud and Prade (2004) in order to assist the agents/DMs to decide about what to say during the dialogue and to support their behavior by founded reasons, namely "safe arguments". This argumentation stage focuses on argumentation-based negotiation dialogues where autonomous agents (representing the DMs) try to find a joint compromise about a collective choice that will satisfy at least all their most important goals, according to their most certain pieces of knowledge.

A general and formal framework for handling these negotiation dialogues is presented here, along with a protocol that specifies rules of interaction among the agents. As the agents negotiate about a set of offers (*i.e. alternatives*) in order to choose the best one from their common point of view, it is assumed that the protocol is run, at most, as many times as there are offers. Since the alternatives used in the Argumentation Stage are the ones provided by the NAI algorithm, the protocol is not run n times, but n^*. Each run of the protocol consists of the discussion of one offer (alternative) by the agents/DMs. If that offer (alternative) is accepted by all the agents/DMs, then the negotiation ends successfully. Otherwise, if at least one agent/DM rejects it strongly and does not revise its beliefs in light of new information, the current offer is – at least temporarily – eliminated and a new one is discussed, initiating a new round.

Two examples are offered to illustrate the function of the entire system, beginning from the setup of the decision problem, using the Ranking Stage to assess the DMs' preferences, then proceeding to identifying negotiable alternatives using the NAI Heuristic and, finally, feeding the results of the NAI Heuristic into the Argumentation Stage for the determination of the commonly acceptable solution. The examples will be presented and explained later on.

II.1.4.1 Mental States and their Dynamics

First of all, a convention needs to be made; for reasons of simplicity and brevity, since each DM is represented by and identified with his/her respected agent, the term agent/DM will be replaced by the term "agent", unless it is considered necessary in a certain context. The mental states of the agents are represented by bases modeling beliefs and goals in terms of certainty and importance respectively. As per Amgoud and Prade (2004), Sierra et al. (1997), each agent is equipped with $2n$ bases, where n is the number of agents participating in the negotiation.

Let L be a propositional language and W$ff(L)$ the set of well-formed formulas built from L. Each agent a_i has the following four bases:

$K_i=\{(k_p^i, \rho_p^i),\ p=1, s_k\}$, where $k_p^i \in Wff(L)$ is a knowledge base gathering the information the agent has about the environment. The beliefs can be less or more certain. They are associated with certainty levels ρ_p^i .

$G_i=\{(g_q^i, \lambda_q^i), q=1, s_g\}$, where $g_q^i \in Wff(L)$ is a base of goals to pursue. These goals can have different priority degrees, represented by λ_q^i . As mentioned earlier, in this system the criteria are identified with the goals.

$GO_j^i=\{(g_{r,j}^i, \gamma_{r,j}^i), r=1, s_{go}(j)\}$, where $j \neq i, go_{r,j}^i \in Wff(L)$, are $(n-1)$ bases containing what the agent a_i believes the goals of the other agents a_j are. Each of these goals has a priority level $\gamma_{r,j}^i$.

$KO_j^i=\{(ko_{t,j}^i, \delta_{t,j}^i), t=1, s_{ko}(j)\}$, where $j \neq i, ko_{t,j}^i \in Wff(L)$, are $(n-1)$ bases containing hat the a_i believes the beliefs of the other agents a_j are. Each of these beliefs has a certainty level $\delta_{t,j}^i$.

The latter case is useful only if the agents intend to simulate the reasoning of the other agents. In negotiation dialogues where agents are trying to reach a common agreement (a consensus), it is more important for each agent to consider the beliefs it has on the other agents' goals (criteria) rather than those of their knowledge. Indeed, a consensus can be more easily reached if the agents ensure that their offers may be consistent with what the believe the goals of the other agents are. So, in what follows, the bases KO_j^i will be omitted. Furthermore, it must be noted that, in this particular system (and, in all likelihood, other systems of this kind), the negotiation process is greatly facilitated by having the same criteria for all DMs (and, consequently, their respective agents).

The different certainty levels and priority degrees are assumed to belong to a unique linearly ordered scale T with maximal element denoted by 1 (corresponding to total certainty and full priority) and a minimal element denoted by 0, corresponding to the complete absence of certainty or priority. m denotes the order-reversing map of the scale. More specifically, $m(0) = 1$ and $m(1) = 0$. The corresponding sets of classical propositions when weights are ignored shall be denoted by K^* and G^*.

II.1.4.2 Argued Decisions

In Amgoud and Prade (2004), a formal framework for decision-making under uncertainty on the bases of arguments that can be built in favor of or against a possible choice was

proposed. This approach has two advantages, which have already been mentioned in earlier sections of this thesis: First, decisions can be more easily explained. Second, argumentation-based decision-making is closer to the way humans make their decisions than approaches which require explicit utility functions and uncertainty distributions. Decisions for an agent are computed from stratified knowledge and preference bases as explained in section II.1.4.1 This approach distinguishes between a *pessimistic* attitude, which focuses on the existence of strong arguments that support a decision, and an *optimistic* one, which focuses on the absence of strong arguments against a decision. This approach can be related to the estimation of qualitative pessimistic and optimistic expected utility measures. Such measures can be obtained from a qualitative plausibility distribution and a qualitative preference profile that can be associated with a stratified knowledge base and a stratified set of goals (Amgoud and Prade 2004). However, in this thesis it is indicated that this approach also works well with measures being provided by a preference disaggregation method (such as any method of the UTA family; in this thesis, the UTASTAR method was chosen).

In this thesis, the syntactic counterpart of these semantical computations is only used in terms of distribution and profile (which has been proven to be equivalent for selecting the best decisions), under its argumentative form.

The idea is that a decision is justified and supported if it leads to the satisfaction of at least the most important goals of the agent, taking into account the most certain part of knowledge. Let D be the set of all possible decisions, where a decision d is a literal. This set D corresponds to the set of alternatives that the agents/DMs are presented with. However, since in this thesis the NAI heuristic is used as an accelerator for the argumentation module, the alternatives in D are *not* the ones in the initial set of alternatives, but instead they are the ones in the *preferred set of alternatives*, as determined by the NAI algorithm.

Definition 1 – Argument PRO: An argument *in favor of* a decicion d is a triple $A = <S, C, d>$ such that:

- $d \in D$
- $S \subseteq K^*$ and $C \subseteq G^*$
- $S \cup \{d\}$ is consistent
- $S \cup \{d\} \vdash C$
- S is minimal and C is maximal (for set inclusion) among the sets satisfying the above conditions.

$S = Support(A)$ is the *support* of the argument, $C = Consequences(A)$ its consequences (the goals which are reached by the decision d) and $d = Conclusion(A)$ is the conclusion of the argument. The set A_P gathers all the arguments that can be constructed from $<K, G, D>$.

Because of the stratification of the bases K_i and G_i, arguments in favor of a decision are more or less strong for i.

Definition 2 – Strength of an Argument PRO: Let $A = <S, C, d>$ be an argument in A_P. The strength of A is a pair $<Level_P(A), Weight_P(A)>$, such that:

- The *certainty level* of the argument is $Level_P(A) = min\{\rho_i \mid k_i \in S \text{ and } (k_i, \rho_i) \in K\}$. If $S = \emptyset$ then $Level_P(A) = 1$.

- The degree of satisfaction of the argument is $Weight_P(A) = m(\beta)$, with $\beta = max\{g_j \mid \lambda_j \in G \text{ and } (g_j, \lambda_j) \notin C\}$. If $\beta = 1$, then $Weight_P(A) = 0$ and if $C = G^*$, then $Weight_P(A) = 1$.

Then, strengths of arguments make the comparison of pairs of arguments possible in the following manner:

Definition 3: Let A and B be two arguments in A_P. A is preferred to B, denoted $A \succeq_P B$, iff $min\{Level_P(A), Weight_P(A)\} \geq min\{Level_P(B), Weight_P(B)\}$.
So, arguments are constructed in favor of decisions and those arguments can be compared. Then decisions can also be compared on the basis of their pertinent arguments.

Definition 4: Let $d, d' \in D$. d is preferred to d', denoted $d \rhd_P d'$, *iff* $\exists A \in A_P$, $Conclusion(A) = d$ such that $\forall B \in A_P$, $Conclusion(B) = d'$, then $A \succeq_P B$.
This decision process is pessimistic in nature since it is based on the idea of ensuring that the important goals are reached. An optimistic attitude can also be modeled. It focuses on the idea that a decision is all the better as long as there is no strong argument against it.

Definition 5 (Argument CON): An *argument against* a decision d is a triple $A = <S, C, d>$ such that:

- $d \in D$
- $S \subseteq K^*$ and $C \subseteq G^*$
- $S \cup \{d\}$ is consistent
- $\forall g_i \in C, S \cup \{d\} \vdash \neg g_i$
- S is minimal and C is maximal (for set inclusion) among the sets satisfying the above conditions.

$S = Support(A)$ is the *support* of the argument, $C = Consequences(A)$ its consequences (the goals which are *not satisfied* by the decision d) and $d = Conclusion(A)$ is the conclusion of the argument. The set A_O gathers all the arguments that can be constructed from $<K, G, D>$.

It must be noted here that the consequences considered here are the negative ones. Again, argument can be more or less strong or weak.

Definition 6 – Weakness of an Argument CON: Let $A = <S, C, d>$ be an argument in A_O. The weakness of A is a pair $<Level_O(A), Weight_O(A)>$, such that:

- The *certainty level* of the argument is $Level_O(A) = m(\varphi)$ such that $\varphi = min\{\rho_i \mid k_i \in S \text{ and } (k_i, \rho_i) \in K\}$. If $S = \emptyset$ then $Level_O(A) = 0$.
- The degree of the argument is $Weight_O(A) = m(\beta)$, such that $\beta = max\{g_j \mid \lambda_j$ such that $g_j \in C \text{ and } (g_j, \lambda_j) \in G\}$.

Once the arguments and their weaknesses have been defined, pairs of arguments can be compared. It is plain to understand that the DM(s) shall prefer decisions that only have weak arguments against them, i.e. what the optimistic attitude is interested in is the least weak arguments against a decision that is being considered. This leads to the following two definitions:

Definition 7: Let A and B be two arguments in A_O. A is preferred to B, denoted $A \gtrsim_O B$, iff $max\{Level_O(A), Weight_O(A)\} \geq max\{Level_O(B), Weight_O(B)\}$.

As in the case of the pessimistic attitude, decisions are compared on the basis of their relevant arguments.

Definition 8: Let $d, d' \in D$. d is preferred to d', denoted $d \rhd_O d'$, *iff* $\exists A \in A_O$, $Conclusion(A) = d$ such that $\forall B \in A_O$ with $Conclusion(B) = d'$, then $A \gtrsim_O B$ (A is preferred to B).

This approach can be illustrated by using the two points of view (pessimistic and optimistic on an example about deciding or not to argue in a multiple agent dialogue for an agent that is not satisfied with the current offer.

Example 1. The knowledge base is $K = \{a \rightarrow suu, 1), (\neg a \rightarrow suu, 1), (a \rightarrow \neg aco, 1), (fco \wedge \neg a \rightarrow aco, 1), (sb, 1), (\neg fco \rightarrow \neg aco, 1), (sb \rightarrow fco, \lambda), (0 < \lambda < 1)$ with the intended meaning:

suu: saying something unpleasant
fco: other agents in favor of current offer
aco: obliged to accept the current offer
a: argue
sb: current offer seems beneficial to the other agents.

The base of goals is $G = \{(\neg aco, 1), (\neg suu, \sigma)\}$ with $(0 < \sigma < 1)$. The agent does not want to say something unpleasant, but it is more important not to be obliged to accept the current offer.

The set of decisions is $D = \{a, \neg a\}$, i.e. arguing or not.

There is one argument in favor of decision "a": $<\{a \rightarrow \neg aco\}, \{\neg aco\}, a>$. There is also a unique argument in favor of the decision "$\neg a$": $<\{\neg a \rightarrow \neg suu\}, \{\neg suu\}, \neg a>$.

The level of the argument $<\{a \rightarrow \neg aco\}, \{\neg aco\}, a>$ is 1, while its weight is $m(\sigma)$. Regarding the argument $<\{\neg a \rightarrow \neg suu\}, \{\neg suu\}, \neg a>$, its level is 1 and its weight is $m(1) = 0$.

The argument $<\{a \rightarrow \neg aco\}, \{\neg aco\}, a>$ is preferred to the argument $<\{\neg a \rightarrow \neg suu\}, \{\neg suu\}, \neg a>$.

From a pessimistic point of view, decision a is preferred to decision $\neg a$ since $<\{a \rightarrow \neg aco\}, \{\neg aco\}, a>$ is preferred to the argument $<\{\neg a \rightarrow \neg suu\}, \{\neg suu\}, \neg a>$.

Now, from an optimistic point of view, there is one argument against decision "a": it is the argument $<\{a \rightarrow suu\}, \{\neg suu\}, a>$. There is also a unique argument *against* the decision "$\neg a$": $<\{sb, sb \rightarrow fco, fco \wedge \neg a \rightarrow aco\}, \{\neg aco\}, \neg a>$.

The level of the argument $<\{a \rightarrow suu\}, \{\neg suu\}, a>$ is 0 whereas its degree is $m(\sigma)$. Regarding the argument $<\{sb, sb \rightarrow fco, fco \wedge \neg a \rightarrow aco\}, \{\neg aco\}, \neg a>$, its level is $m(\lambda)$ and its degree is 0. Then the comparison of the two arguments comes down to comparing $m(\sigma)$ to $m(\lambda)$.

This comparison is what will determine the final recommended decision when using the optimistic approach.

This argumentation system will be used to make decisions about the offers to propose in a negotiation dialogue. The following definition is the same as Definition 1, but here the decision is about *offers*.

Definition 9 (Argument for an Offer): An argument *in favor of* an offer x is a triple $A = <S, C, d>$ such that:

- $x \in X$
- $S \subseteq K^*$ and $C \subseteq G^*$
- $S(x)$ is consistent
- $S(x) \vdash C(x)$
- S is minimal and C is maximal (for set inclusion) among the sets satisfying the above conditions.

X is the set of offers, $S = Support(A)$ is the *support* of the argument, $C = Consequences(A)$ its consequences (the goals which are reached by the offer x) and $x = Conclusion(A)$ is the conclusion of the argument. $S(x)$ – and, respectively, $C(x)$ – denotes the belief state – and, respectively, the preference state – when an offer x is proposed.

Example 2. This example presents the case of an agent who wants to propose an offer corresponding to its desired holiday destination. The set of available offers is $X = \{Tunisia, Italy\}$.

The knowledge base is $K = \{(Sunny(Tunisia), 1), (\neg Cheap(Italy), \beta), (Sunny(x) \rightarrow Cheap(x), 1)\}$

The preferences base is: $G = \{(Cheap(x), 1)\}$

The decision to be made by the agent is whether Tunisia or Italy should be offered. Following the last definition, it has an argument in favor of Tunisia: $A = <\{Sunny(Tunisia), Sunny(x) \rightarrow Cheap(x)\}, Cheap(Tunisia), Tunisia>$

There is no argument in favor of Italy, as it violates the agent's goal, which is very important. So, the agent will offer Tunisia.

II.1.4.3 The Negotiation Protocol

II.1.4.3.1 Formal setting

In this section, the formal protocol for the handling of negotiation dialogues among many ($n \geq 2$) agents is presented. The agents have to discuss several offers, so the protocol will be run as many times as there are non-discussed offers and such that a common agreement has not yet been found. The agents take turns to start new runs of the protocol and only one offer is discussed during each turn.

A negotiation interaction protocol is a tuple ⟨ Objective, Agents, Object, Acts, Replies, Wff-Moves, Dialogue, Result ⟩ such that:

Objective is the aim of the dialogue, namely to find an acceptable offer.

Agents is the set of agents participating in the dialogue, $Ag = \{a_0, \ldots, a_{n-1}\}$.

Object is the subject of the dialogue. It is a multi-issue dialogue, denoted by the tuple $\langle O_1, \ldots, O_m \rangle$, $m \geq 1$. Each O_i is a variable taking its values in a set T_i. Let X be the set of all possible offers. Its elements are $x = \langle x_1, \ldots, x_m \rangle$, with $x_i \in T_i$..

Acts is the set of possible negotiation speech acts: *Acts* = {Offer, Challenge, Argue, Accept, Refuse, Withdraw, Say nothing}.

Replies: Acts → Power(Acts) is a mapping that associates each speech act to its possible replies:

- *Replies*(*Offer*) = {Accept, Refuse, Challenge}
- *Replies*(*Challenge*) = {Argue}
- *Replies*(*Accept*) = {Accept, Challenge, Argue, Withdraw}
- *Replies*(*Refuse*) = {Accept, Challenge, Argue, Withdraw}
- *Replies*(*Withdraw*) = ∅

Well-founded moves (Wff-moves): $\{M_0, \ldots, M_p\}$ is a set of tuples $M_k = \langle S_k, H_k, Move_k \rangle$, such that:

- $S_k \in Ag$, the agent which plays the move is given by the function $Speaker(M_k) = S_k$.
- $H_k \in Ag \setminus \{S_k\}$, the set of agents to which the move is addressed is given by the function $Hearer(M_k) = S_k$.
- $Move_k = Act_k(c_k)$ is the uttered move where Act_k is a speech act applied to a content c_k.

Dialogue is a finite non-empty sequence of well-founded moves $D = \{M_0, \ldots, M_p\}$ such that:

- $M_0 = \langle S_0, H_0, offer(x) \rangle$: each dialogue starts with an offer $x \in X$.
- $Move_k \neq offer(x)$, $\forall\ k \neq 0$ and $x \in X$: only one offer is proposed during the dialogue at the first move.

- $Speaker(M_k) = a_{k \ modulo \ n}$: the agents take turns during the dialogue.
- $\text{Speaker}(M_k) \notin Hearer(M_k)$: This condition forbids an agent to address a move to itself.
- $Hearer(M_0) = a_j, \forall j \neq i$: the agent a_i which performs the first move addresses it to all the agents.
- For each pair of tuples M_k, M_h, $k \neq h$, if $S_k = S_h$ then $Move_k \neq Move_h$. This condition forbids an agent to repeat a move that has already been performed.

These conditions guarantee the ***non-circularity*** of the dialogue D, i.e. that the dialogue will not repeat the same moves.

Result: $D \rightarrow \{success, failure\}$ is a mapping which returns the result of the dialogue.

- $Result(D) = success$ if the preferences of the agents are satisfied by the current offer.
- $Result(D) = failure$ if the most important preferences of at least one agent are violated by the current offer.

This protocol is based on dialogue games. Each agent is equipped with a *commitment store* (CS) [MacKenzie, (1979)] that contains the set of facts it is committed to during the dialogue. Using the idea introduced in [Amgoud, Maudet and Parsons, (2002)] of decomposing the agents' commitment store (CS) into many components, it is supposed that each agent's CS has the structure:

$$CS = \langle S, A, C \rangle$$

with:

$CS.S$ containing the offers proposed by the agent and those it has accepted ($CS.S \subseteq X$).
$CS.A$ being the set of arguments presented by the agent ($CS.A \subseteq Arg(L)$), where $Arg(L)$ is the set of all arguments that can be constructed from L.
$CS.C$ being the set of challenges made by the agent.

At the first run of the protocol, all the components of the CS are empty. This is not the case, of course, when the protocol is run again, because agents need to keep their previous commitments in order to avoid repeating what they have already said and done during the previous runs of the protocol (this ensures the non-circularity of the dialogue).

II.1.4.4 Conditions on the negotiation acts

Here, the pre-conditions and post-conditions (effects) for each act will be specified. For the agents' commitments (CS), only the changes to effect are specified. It is supposed that the agent a_i addresses a move to the $(n - 1)$ other agents.

Offer(x) where $x \in X$. The idea is that an agent chooses an offer x for which there are the strongest supporting arguments w.r.t. G_i. Since the agent is *cooperative* (meaning that it

tries to satisfy its own goals taking into account the goals of the other agents), this offer x is also the one for which no strong argument against it exists (using GO^i_j instead of G_i).

Pre-conditions: Among the elements of X, choose x which is preferred to any $x' \in X$ such that $x \neq x'$, in the sense of definition 4, provided that there is no strong argument this offer (i.e. with a weakness degree equal to 0) where G_i is changed into GO^i_j , $\forall j \neq i$ in definition 8.

Post-conditions: $CS.S_t(a_i) = CS.S_{t-1}(a_i) \cup \{x\}$.

Challenge(x) where $x \in X$. This move incites the agent which receives it to give an argument in favor of the offer x. An agent asks for an argument when this offer is not acceptable for it and it knows that there are still non-rejected offers.

Pre-conditions: $\exists\, x' \in X$ such that x' is preferred to x w.r.t definition 4.

Post-conditions: $CS.C_t(a_i) = CS.C_{t-1}(a_i) \cup \{x\}$: The agent a_i which played the move *Challenge*(x) keeps it in its CS.

Challenge(y) where $y \in Wff(L)$. This move incites the agent which receives it to give an argument in favor of the proposition y.

Pre-conditions: None.

Post-conditions: $CS.C_t(a_i) = CS.C_{t-1}(a_i) \cup \{y\}$: The agent a_i which played the move *Challenge*(y) keeps it in its CS.

Argue(S) with $S = \{(k_p, a_p), p = 1, s\} \subseteq K_i$ is a set of formulas representing the support of an argument given by agent a_i.

Pre-conditions: S is acceptable.

Post-conditions: $CS.S_t(a_i) = CS.S_{t-1}(a_i) \cup S$. If S is acceptable, the agents a_j *revise their base* K_j into a new base $(K_j)^*(S)$.

Withdraw An agent may withdraw from the negotiation if it does not have an acceptable offer to propose.

Pre-conditions: $\forall\, x \in X$, there is an argument with maximal strength against x, or $(X = \varnothing)$.

Post-conditions: *(Result*(D) = *failure)* and $\forall\, i$, $CS_t(a_i) = \varnothing$. As soon as an agent withdraws, the negotiation ends and all the commitment stores are emptied. The dialogue is presumed to end this way because the aim is to find a compromise between the n agents participating in the negotiation.

Accept(x) where $x \in X$. This move is performed when the offer x is acceptable for the agent.

Pre-conditions: The offer x is the most preferred decision in X in the sense of definition 4.

Post-conditions: $CS.S_t(a_i) = CS.S_{t-1}(a_i) \cup \{x\}$. If $x \in CS.S(a_i)$, $\forall\ i$, then $Result(D) = success$, i.e. if all the agents accept the offer x, the negotiation ends with x being the compromise.

Accept(S) $S \subset Wff(L)$.

Pre-conditions: S is acceptable for a_i.

Post-conditions: $CS.A_t(a_i) = CS.A_{t-1}(a_i) \cup S$.

Refuse(x) where $x \in X$. An agent refuses an offer if it is does not consider it acceptable.

Pre-conditions: There exists an argument in the sense of definition 5 against x.

Post-conditions: If $\forall\ a_j$, $\nexists\ (S, x)$, i.e. if no acceptable argument for x exists then $X = X \setminus \{x\}$. A rejected offer is removed from the set X. $Result(D) = failure$.

Say nothing: This move allows an agent to skip its turn if it has already accepted the current offer or if it has no argument to present. This move has no effect on the dialogue.

II.1.4.5 Properties of the negotiation protocol

Property 1: Termination. Any negotiation among n agents managed by this protocol ends, either with $Result(D) = success$ or $Result(D) = failure$.

Property 2: Optimal Outcome. If the agents do not misrepresent the preferences of the other agents (GO^i_j), then the compromise found is an offer x which is preferred to any other offer $x' \in X$ in the sense of definition 4, for all the agents.

II.1.5 Innovations, challenges and changes in the proposed methodology and software

Admittedly, in an area where many researchers try to present new advances every year, it is not particularly easy to find a niche that provides enough room for innovation compared to what other scientists have achieved and/or are working on at the specific time. However, the combination of, and improvement upon, existing techniques is promising, as it provides a way to enhance the performance of a given set of methodologies and contribute to the creation of an approach that combines the advantages of each method used, while allowing for the avoidance of their shortcomings.

As mentioned before in this thesis, three techniques are combined for the solution of the choice problem that the group of DMs are facing; the UTASTAR preference disaggregation method, which provides each DM's individual ranking of the alternatives, as

well as the utilities (marginal and global) of the alternatives and the weights of the criteria for each DM; the NAI algorithm, which takes over from the UTASTAR method's results and provides a very elegant, efficient and, above all, fast way of identifying a set of alternatives that the DMs would consider negotiating upon. The NAI algorithm has the potential of greatly reducing the set of alternatives upon which the agents representing the DMs will negotiate, therefore considerably accelerating the negotiation process; indeed, the *n* agents will now deliberate not on the *n* initial alternatives (the original set of alternatives *A*), but on (*n* – *n'*) instead (the *preferable* subset of alternatives). Finally, the argumentation-based negotiation protocol, which enables the autonomous agents that represent the DMs to interact with each other, present their positions, argue on them, justify them, change them, so that eventually a common ground, a consensus, a compromise can be reached that will satisfy the agents' most important goals – and all this in a manner that will be more easily comprehensible by the DMs, *i.e.* the users of the system; the users are not necessarily experts in the Decision Science and the developer of a DSS should never assume they are. Furthermore, to break an intersection impasse (if it occurs), the algorithm incorporates procedures 1 and 2 in order to ensure that the most preferable and the preferable subset of alternatives are non-empty. Thus, the *Argumentation Stage* that follows is guaranteed to be provided with alternatives for the agents to negotiate on.

The argumentation-based negotiation framework proposed by Amgoud, Belabbes and Prade (2005) that is adapted in this thesis can use arguments expressed as functions of numerical (quantitative or quantified qualitative) data acquired either ready from a database that contains them or as results of the application of a multicriteria decision analysis method. The basic idea of the formal decision framework used by the agents in the *Argumentation Stage* is that an agent utters and accepts offers that are supported by strong arguments. Similarly, agents can refuse or challenge offers against which there is at least one strong argument. The protocol for the interaction among the agents is run at most as many times as there are non-discussed offers; at each run only one offer is discussed. If all the agents accept it, then an agreement has been found. If not, it is removed from the set of offers and another one is proposed

The case presented in this thesis is the latter: the arguments are expressed as functions of the alternatives' utilities and the weights of the criteria, which is a departure from the original protocol's typical approach; Amgoud, Belabbes and Prade (2005) express the arguments as functions of the alternatives' *performances* in the criteria; this greatly reduces an argumentation protocol's ability to take into account the subjective preferences of a DM, as it only takes into account the *objective* value of an alternative in a specific criterion. For instance, if the alternatives were cars, the original protocol would only take into account objective values in criteria such as price, fuel consumption, top speed, luggage space, engine displacement, horsepower, torque etc; it would, therefore, only take into account what is known as the *specifications* and *features* of these alternatives. It would not cater for problems where the alternatives must be judged (and negotiated upon) based on *what they are worth* for the DMs.

The capability of the argumentation-based negotiation framework to easily handle numerical data and build its arguments as functions of numerical values makes it especially suitable for solving such decision problems as the ones for which the methodology presented in this thesis was developed: indeed, the results of the UTASTAR method can be used immediately as input for this argumentation framework, without the need for any significant adaptation. It also makes it especially suitable for implementing in languages

other than languages dedicated to Artificial Intelligence; like the rest of the protocols presented by Amgoud et al., it lends itself very well to programming in Java, C, C++ and other such languages.

However, the protocol is not without its weaknesses: it could be more flexible if it stored the rejected offers in stratified sets, calculating levels of rejection (perhaps in the same vein as the structural index of preferences of the NAI algorithm): in the same manner as the NAI algorithm's least preferable subset, the last set of such a stratification would gather the offers that are definitely rejected. Once all the offers were studied without finding an acceptable one, the agents would negotiate again on the set gathering the less rejected offers and proceed in the same way, having revised their bases and becoming less demanding w.r.t their preferences. Such a development of the protocol, however, is similar to what is achieved in this thesis through the application of the NAI heuristic; this is not to say, however, that an investigation of how such an enhanced version of the argumentation protocol, combined with the already implemented techniques in this methodology and its accompanying software application, could improve upon the current system lacks merit. Furthermore, more advanced principles for the comparison of arguments (for instance, principles combining a *Prevention focus* and a *Promotion Focus* (Amgoud, Bonnefon and Prade 2005) could be implemented in later revisions of the methodology proposed here.

In essence, the proposed methodology and software combines the advantages of the NAI algorithm (speed, ease of coding, intuitive usage) with those of argumentation-based negotiation (ability for a DM to explain, justify and, if necessary, change his/her position) to enable a group of DMs to reach a consensus on a commonly accepted solution to a choice problem, using the NAI algorithm as an accelerator for the *Argumentation Stage*, avoiding, at the same time, the disadvantage of the initial argumentation protocol modeling the decision problem based on the alternatives' performances in the various criteria instead of their marginal utilities. The usage of the NAI heuristic algorithm for the determination of an initial region in which a possible solution can be located is not unlike the use of heuristics in antivirus software applications (such as Carey S. Nachenberg's patent no. 6357008 “Dynamic Heuristic Method for Detecting Computer Viruses Using Decryption Exploration and Evaluation Phases”, applied on behalf of Symantec Corporation on September 23, 1997).

The methodology set forth in this thesis seems like a combination of already existing techniques. However, in this combination lie its main scientific contributions:

1. The usage of a heuristic algorithm as a method for accelerating the argumentation-based negotiation taking place among the agents. The argumentation process is accelerated by the heuristic algorithm through the narrowing down of the area in which the agents will search for a solution. This narrowing down is achieved by cutting off the alternatives that the agents consider undesirable and therefore reducing the number of alternatives on which the agents will deliberate.
2. The argumentation protocol proposed by Amgoud, Belabbes and Prade (2005) was improved by using the *marginal utilities* of the alternatives instead of their performances in the various criteria. This change improves the protocol's ability to take each agent's subjective stance towards the problem into account; as the cardinal value used in the formulation of an argument now reflects exactly what an agent thinks of an alternative (by using the marginal utility of the alternative, by

definition a subjective measure) instead of how well the alternative performs w.r.t. a specific criterion (which is an objective value).

3. Another measure taken to ensure that the argumentation protocol would express the DMs' individual subjective perception of the problem and their personal beliefs is the use of each individual DM's weights for the criteria (goals).

At the time this methodology was being developed, applications based on the methodologies and techniques that were used as its components were few and not always easily available or even usable; teaching experience with the MINORA and MIIDAS systems at the Technical Educational Institute of Larissa has uncovered compatibility issues with recent versions of Windows (Vista and 7, especially 64-bit); their source code was not available for study, update and modification and, even if it was, since the entire software that was developed as part of this thesis was to be written in the Java programming language, the effort to rewrite the code from one language to another would add unwanted workload and possible delays. Furthermore, at the time there were practically no software classes and libraries for UTASTAR in Java that such a system could be based on. The most commonly software implementation of the UTASTAR method was one using MATLAB, which unfortunately could not be considered as an option, since it would impose dependency on an external – and expensive – piece of software, which could make the software unattractive to cost-conscious standalone users, such as very small enterprises. Indeed, if an implementation of UTASTAR dependent on MATLAB (or any other proprietary/commercial scientific suite, such as Mathematica) was used, a potential user would be forced to spend money on a piece of software that perhaps s/he does not have reasons to use elsewhere. Furthermore, there are often interoperability issues between different applications that can hamper the development effort for such a system; another factor that weighed in against the use of this implementation was the ambition to further develop this system for use in mobile devices in the future (netbooks, smartphones and, nowadays, tablet PCs). Dependence on an application that is unavailable on widespread mobile platforms was undesirable.

Likewise, there was no known software library implementing the NAI algorithm in any programming language. Either it had been implemented only as part of a dedicated spreadsheet file or whatever software routines might have been written were entirely proprietary and closed source applications that might have been *ad hoc* and not reusable or available for other researchers to use in their own works. However, this algorithm is fairly straightforward to code.

Other than implementing UTASTAR in Java, the greatest challenge was to write an argumentation-based negotiation protocol in a programming language other than a language dedicated to Artificial Intelligence (such as Prolog). Despite the existence of the JADE (Java Agent Development Engine) platform, which has recently also been implemented as a plug-in for the Eclipse IDE, the decision was made for the development of the argumentation framework and protocol without the use of JADE, as the IDE that was chosen from the beginning of the development of this software was Oracle's Netbeans and work had already progressed rather significantly. Thus, the switch to Eclipse and its JADE plug-in named EJADE was not an attractive option at the time, as it would impose the cost in time and effort of dealing with an extra learning curve, although it has already been scheduled for future work.

Chapter III
THE PROPOSED SOFTWARE

In this chapter, the proposed software is presented. The following issues are detailed and documented: (a) requirements that the software must satisfy (b) the way it was implemented in order to satisfy these requirements, (c) documentation for its usage using two example problems.

III.1 Requirements and brief for the software

As set forth in the goals and scope of this thesis, the aim of the proposed methodology and software is to support a group of DMs in solving a *choice* problem under multiple criteria and in a collaborative context; i.e. a group of DMs who want to cooperate in order to *choose* one action (alternative) from a set of many. To solve this problem, the proposed methodology combines the UTASTAR method (Siskos and Yannacopoulos 1985) in order to estimate the individual preferences of the DMs that make up the group (the *Ranking Stage* explained and detailed in Chapter II). That way, the software creates individual ordinal rankings for all the DMs of the group. In the second stage (the *Identification of Negotiable Alternatives Stage* presented in detail in Chapter II), the NAI algorithm is applied to the ordinal rankings of the DMs in order to acquire a new, smaller set of alternatives that the DMs would be willing to consider, even if they are not at the top of their respective lists. Finally, in the third stage (the *Argumentation Stage* detailed in Chapter II), arguments *for (pro)* and *against (con)* each alternative from the set created by the NAI algorithm in the second stage are formulated and compared. This last stage is the one that eventually leads to the choice of *one* solution. Other requirements for the proposed software are:

1. Ease of use even for DMs unfamiliar with MCDA techniques.
2. Production of accurate results with as little computational error as possible.
3. Use of the Java programming language for portability to numerous different platforms (Windows, GNU/Linux and UNIX-based systems, Mac OS X, iOS, Google Android etc); i.e. the software needs to be platform-agnostic.
4. Complete independence from proprietary and potentially expensive software, especially software this is not available on every platform that the system might be further developed for; the software must be entirely self-contained.
5. Modular structure that will facilitate the improvement/refinement of existing features and capabilities and the addition of new ones.

6. It must form a core for later development and evolution of the system into a full-featured distributed multi-agent, multi-user system that will also feature:
    a. Asynchronous cooperation among the DMs, *i.e.* the DMs over the internet or on a local network (a Wi-Fi network or a Local Area Network, for instance).
    b. A client-server architecture, in which the clients will act in the same capacity as agents in a multi-agent system and the server will handle the entire computational, coordination and central data storage load (with the option of local data storage as well) *or*, alternatively,
    c. A peer-to-peer architecture where each DM's client-agent will also handle his/her individual UTASTAR calculations, while the calculations of the NAI algorithm could be handled by the agent of the DM that initiates the session; the argumentation will be handled by each DM's client-agent.

III.1.1 Implemented features

In the form that is presented here as part of this thesis, the software developed and supplied here is implemented entirely in Java and provides an intuitive, easy-to-use Graphical User Interface (GUI) that, in the same vein as wizard-style applications (much like the installers that are now popular with many applications and operating systems), guides the user from one step to the next; it also allows the user to go back to a previous step in order to correct or modify data s/he has entered. Networking capabilities, which are crucial for the software's future stages of development, in which the system will evolve into a complete multi-agent, distributed, asynchronous collaborative GDSS, where argumentation will be handled by independent intelligent agents for each user/DM, have not yet been implemented because the main focus was placed on implementing the UTASTAR method in the Java programming language and its combination with the NAI heuristic algorithm and an argumentation protocol, while ensuring the validity of the method and the results produced by the system.

Like the ranking stage, the argumentation stage has been implemented in Java, although languages specifically created for Artificial Intelligence and logic applications, such as Prolog, could perhaps have been better suited for this particular task. However, it was decided from the very beginning of this undertaking that the entire system would be written in a single programming language, namely Java. Furthermore, the portability enjoyed by Java applications, as well as the fact that it has become a popular programming language with good community support added to its appeal.

As is the case with any Java application, the proposed software uses *classes* and *libraries* and is inherently modular. This allows for easy debugging and improvements and also for the addition of further features and capabilities in future versions.

III.1.2 Functions of the proposed software

As has been mentioned before, the proposed system combines the UTASTAR method with the NAI algorithm and an argumentation framework. The UTASTAR method is applied in order to:

- Calculate the individual marginal and global utility values *for the entire group of DMs, i.e.* the marginal and global utilities of the alternatives for each individual DM.
- Calculate the weights of the criteria for each individual DM.
- Calculate the individual DMs' rankings of the alternatives.

After this stage, the NAI algorithm is employed to identify alternatives that the DMs would consider negotiating upon, even if they are not at the top of their list. The NAI algorithm is a three-step process that provides three subsets of alternatives, common for all the DMs: the most preferable subset of alternatives, the more preferable subset of alternatives and the least preferable subset. The most important subset produced by the NAI algorithm is the more preferable subset, as it is this subset that will provide the list of alternatives upon which the agents representing the DMs will negotiate.

After the negotiable alternatives have been identified, the *Argumentation Stage* set forth in Chapter II is applied so that *one* alternative will be eventually proposed to the group of DMs. Although it is intuitive for one to argue that the most preferable alternative, according to the ranking of the alternatives, would be the one that is finally chosen, there are occasions where this does not apply. In group decision making contexts, the ways the DMs think and their priorities may vary (and often do) widely. This variation is reflected in the DMs' preferences, as expressed by the individual rankings of the alternatives. In such cases, a compromise must be reached. This is where negotiations (either game-theoretic or argumentation-based) are used. However, in problems where the number of DMs and/or the number of alternatives is very large, the negotiation (regardless of whether it is game-theoretic or argumentation-based) can take a long time to complete.

It is for this reason that the NAI heuristic is used; the NAI algorithm, by identifying alternatives that the DMs view as interchangeable and worth considering instead of the ones that are at the top of their lists, saves time from the negotiation process, as it reduces (often significantly) the number of alternatives upon which the DMs (or, more suitably for methodologies and software applications such as the one presented here), will deliberate in order to reach a consensus.

The argumentation stage of the methodology and the software that implements it is based on a general formal framework for dialogue among autonomous agents that seek a common agreement about a collective choice. The setting has three main components: the agents, their reasoning capabilities and a protocol. The agents are supposed to maintain beliefs about the environment and the other agents, along with their own goals.

III.2 Structure of the proposed system

The proposed software, in keeping with the methodology it implements, consists of three main parts:

1. A *Ranking Stage* (which is an implementation of the UTASTAR algorithm in Java). In this stage, the program solves a UTASTAR problem for each DM, calculating the global and marginal utilities of the alternatives for each DM. This is implemented by the class com.gorbas.UTASTAR.
2. The *NAI algorithm*, in which the program determines a set of negotiable alternatives, which can be different from the ones that each DM considers most preferable and places at the top of his/her ranking. This is implemented in the class com.gorbas.NAI. The NAI algorithm is run for each DM in the form of the class com.gorbas.NAI.DecisionMakerContext, in which the expansion and contraction operations are performed. The method NAI.run() runs the algorithm for all DMs and, in case there is an intersection impasse, applies the necessary procedures to break it.
3. The *Argumentation Stage* , in which the agents representing the DMs engage in an argumentation-based negotiation dialogue aiming to reach a consensus on a commonly acceptable choice.

To achieve its goals, the application creates a database in which it stores the data it needs in the form of tables:

1. The criteria of the decision problem
2. The decision makers
3. The alternatives and their performances in each criterion
4. The individual rankings of the alternatives as determined by the UTASTAR method
5. The structural indices of preferences, cut-off points and the subsets calculated by the NAI algorithm

The argumentation-based negotiation protocol is formed in two parts. In the first part, its elements are formed using specific classes for each:

1. The *Agents*, which are the most complex entities of the protocol: they respond to the impulses given to them by their environment in the manners described in Chapter 2; they make, accept or refuse offers and provide arguments when it is demanded of them.
2. The *Moves:* there is a specific subclass for each type of move described in the protocol, each with its appropriate properties. All the moves that can be challenged belong in the *Move.Challengable* class.
3. *Act*. This indicates the type of the Move, to avoid the need for constantly checking it.
4. *Dialogue:* Represents one round (run) of the protocol; it contains the moves made and the outcome.
5. *Offer:* This is an offer made by an agent that starts a run of the protocol in the dialogue; in the case of this system, an offer is identified with an alternative from the set of preferred alternatives provided by the NAI algorithm.
6. *Argument:* The arguments in this implementation of the protocol are expressed as functions of the utilities and criteria weights calculated by the UTASTAR method.

It must be noted that the *Goals* in this implementation of the protocol are practically embedded in the notion of an argument. This is possible thanks to the following facts: (a)

the goals can easily be identified with the criteria of the UTASTAR method, (b) the protocol itself refers to the goals only when the arguments are created and assessed.

The second part of the argumentation-based negotiation protocol, implemented by the com.gorbas.negotiation.amgoud.impl.ProtoImpl class and its internal classes, specifies the aforementioned elements by using the output of the UTASTAR method and the preferred subset of alternatives provided by the NAI algorithm in the following manner:

- ProtoImpl.DmOffer: This class presents an alternative as an offer in the protocol.
- ProtocolImpl.DmArgument: An argument as a function of the marginal utility of an alternative on a certain criterion (goal). In this class, the comparison of the arguments also takes place.
- ProtoImpl.Agent: This class presents a DM (DecisionMaker) as an agent in the protocol and creates arguments using the marginal utilities of each alternative on each criterion as computed by the UTASTAR method.

Figure 3 (on page 54) presents the structure of the proposed system.

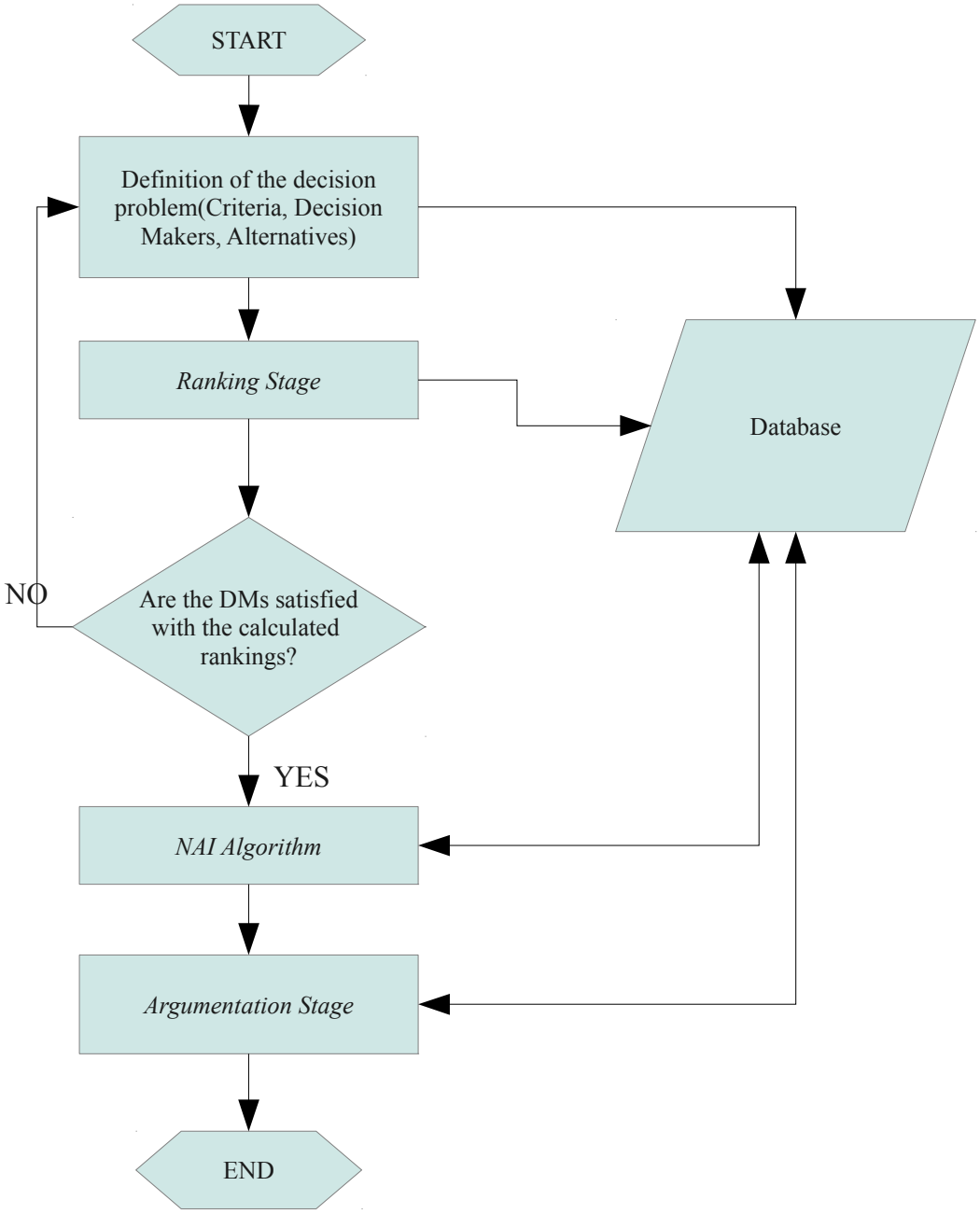


Figure 3. Structure of the proposed system.

III.3 Using the proposed system

First of all, it must be noted that the proposed software does not require any installation; the user only needs to copy the folders with the necessary files to whatever directory s/he chooses.

III.3.1 Starting the application

The complete filename for the application is PHD-jar-with-dependencies.jar. As mentioned above, no installation is required; it is absolutely self-contained and the only requirement is the existence of version 6 (or more recent) of Sun's Java Runtime. In the example shown in Figure 4, one can see the icon of the application in the folder where it was placed (the case depicted is a laptop running Ubuntu Linux 10.10 with the Gnome 2.32 desktop environment and Nautilus as the file manager; further tests were made with the same machine running Ubuntu Linux 12.04 with the Unity desktop environment). The **db_u** subfolder in the file manager window is the folder where the application stores the files it creates; these files contain the decision problems created by the user.

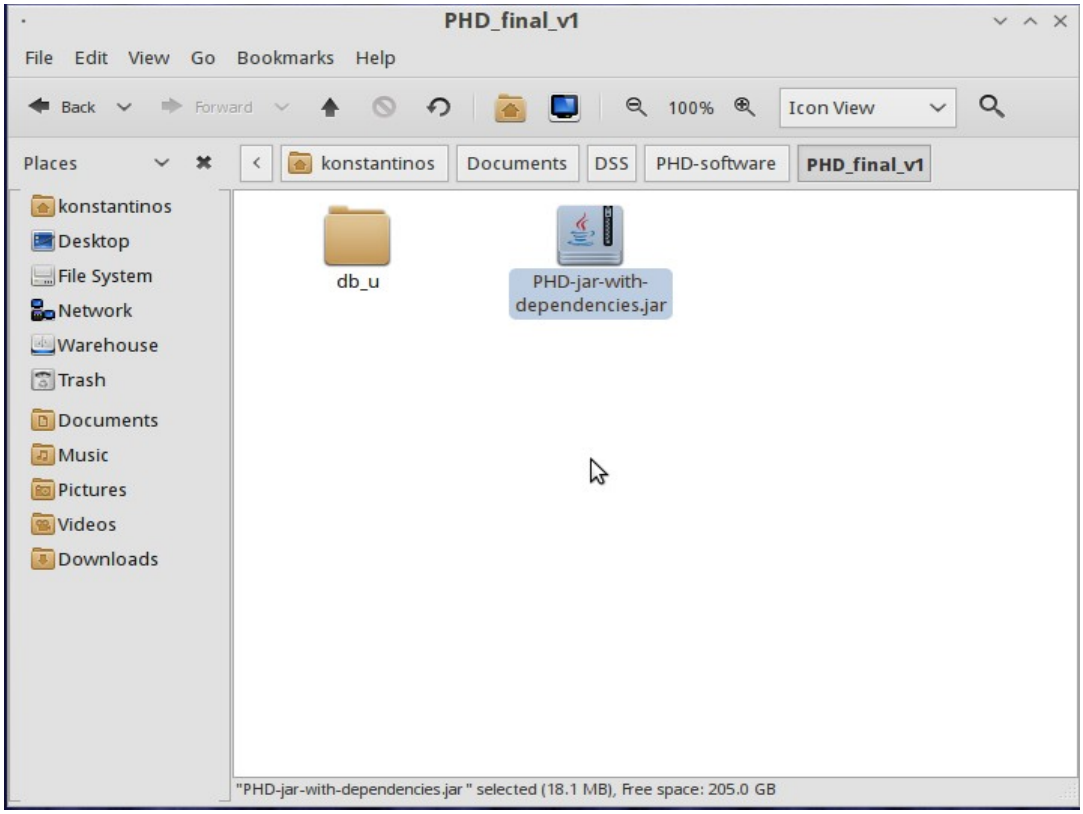


Figure 4. The application located in the folder where it was copied.

In the case of a UNIX or GNU/Linux-based system, the application can be run by *right-clicking* on the icon (*not* left-clicking) and selecting "Run with OpenJDK Java 6 Runtime" from the context menu (Figure 5, on the next page). On Windows-based machines, the procedure is similar; the user needs only to specify which application (the Java 6 Runtime) will be used to run the application. Mac OS X and iOS-based machines were not available at the time the application was being developed, but starting the application on them should not be considerably different. After all, the application uses a cross-platform language to run and so whatever differences are entirely a matter of how each different operating system handles applications that need an interpreter and have not been compiled as executables.

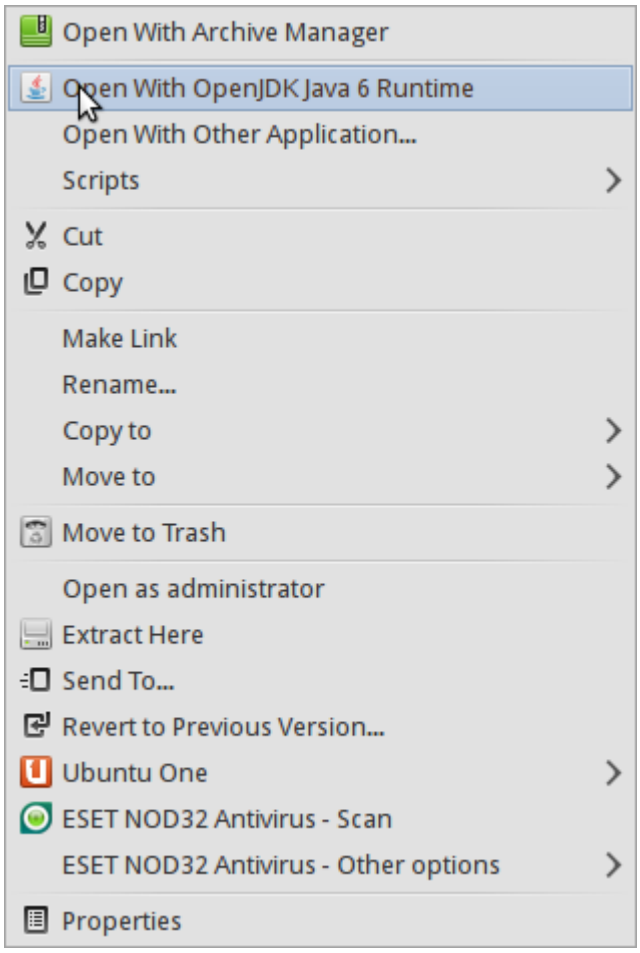


Figure 5. Opening the application.

Upon opening the application, the user is presented with a window prompting him/her to enter the name of the contest (i.e. the decision-making session/decision problem in which the DMs will have to choose a certain action from a set of alternatives). Providing a name for the contest is not necessary, but it helps if the DMs wish to recall the problem from the application's database at a later date for future reference.

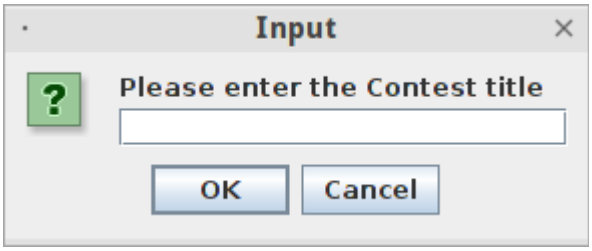


Figure 6. Naming the decision problem.

As explained before, the user may or may not choose to give a name to the decision problem. For instance, if the user is planning to open an existing problem, s/he may click on “OK” or “Cancel” without providing a name for the problem; whatever his/her action is at this stage is irrelevant. In this chapter, both the case where the user opens an existing

problem and the case in which the user creates a decision problem from scratch will be examined.

III.3.2 Creating a new decision problem

First, the case where a new problem is created from scratch is presented. It will be an extension of the transport medium selection problem presented in [Siskos and Yannacopoulos (1985)]. Here, the problem will have two DMs; each one will rank the alternatives in a different manner compared to his/her counterpart. So, the problem will be as follows:

Consider the case of two persons living in Paris, wishing to choose the most suitable transport to go to their workplace; these two persons are colleagues, working for the same business and they also happen to live in the same neighborhood; thus, using the same transport for commuting to their work is convenient and sensible. They have the following means of transport at their disposal: A = {RER, METRO-1, METRO-2, BUS, TAXI}, which will be assessed using the following three criteria: Price (quantitative criterion, measured in Francs - Fr), duration of journey (quantitative, measured in minutes – min) and comfort (chance of finding an empty seat – this is a qualitative criterion).

The qualitative criterion "Comfort" is quantified using the following scale:

0	No vacant seat
+ (or 1)	Low probability of finding an empty seat
++ (or 2)	High probability of finding an empty seat
+++ (or 3)	Empty seat assured

Table 2. The quantification of the qualitative criterion "comfort".

Now, the two DMs will have to express their preferences for each alternative on the criteria used in this problem. Finally, they will have to express their overall preferences on the alternatives, ranking them from the most preferred to the least preferred.

DM 1 (assume that his name is Antoine) has the following preferences:

$$\text{RER} \sim (\text{METRO-1} \succ \text{METRO-2}) \succ \text{BUS} \succ \text{TAXI}$$

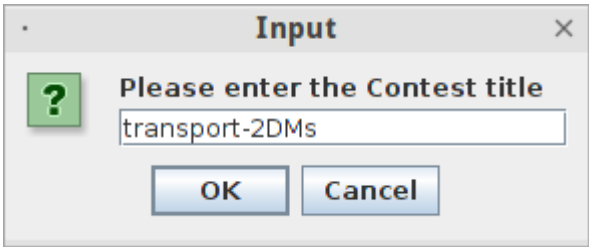
While DM 2 (assume that his name is Gregoire) expresses the following preferences:

$$\text{RER} \succ \text{METRO-1} \succ \text{METRO-2} \succ \text{BUS} \succ \text{TAXI}$$

This leads to the formation of the following multicriteria table (Table 3):

Transport types	Price (Fr)	Duration of Journey (min)	Comfort	Weak order (DM 1)	Weak order (DM2)
RER	g_1(RER) = 3	g_2(RER) = 10	g_3(RER) = +	1	1
METRO-1	g_1(METRO-1) = 4	g_2(METRO-1) = 20	g_3(METRO-1) = + +	2	2
METRO-2	g_1(METRO-2) = 2	g_2(METRO-2) =20	g_3(METRO-1) = 0	2	3
BUS	g_1(BUS) = 6	g_2(BUS) = 40	g_3(BUS) = 0	3	4
TAXI	g_1(TAXI) = 30	g_2(TAXI) = 30	g_3(TAXI) = +++	4	5

Table 3. Multicriteria table for the two-DM transport choice problem



The name given to the problem ("Contest" in the application) is "transport-2DMs".

Figure 7. Naming the new decision problem

Then, the user simply has to click "OK". Then, s/he is presented with the screen of Figure 8:

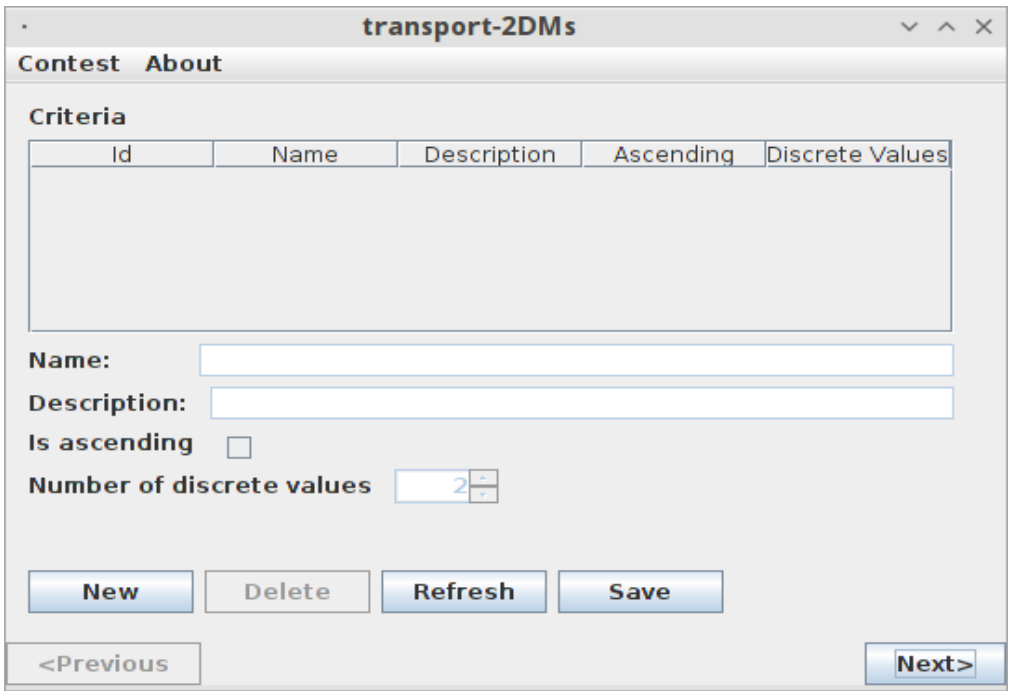


Figure 8: Criteria entry

On this screen, the user may enter the criteria of the decision problem s/he wishes to create. On the title bar of the window, the name of the decision problem is visible. Underneath, the "Contest" option allows the user to start a new problem or load an existing one. Below, a table containing the criteria is created as the user enters the criteria used for the problem.

Each criterion has its own name; optionally, it may have a description as well, although this is not needed. If the criterion is ascending, the user has to tick the relevant option. Also, the user has to enter the number of discrete values for each criterion; these discrete values are the ones used by the UTASTAR method. Then are the buttons "New", "Delete" (deactivated if no entries have been made), "Refresh" and "Save". Most of these buttons are self-explanatory; "New" creates a new criterion. "Delete" deletes a selected criterion; "Save" stores the problem in its current state; finally, the "Refresh" button is used to overcome a bug that sometimes does not allow the application to read the data and perform the calculations as it should. Finally, there are the "Next" and "Previous" buttons typical of all wizard-style applications.

To create a new criterion, the user must click on "New" and is then presented with the following:

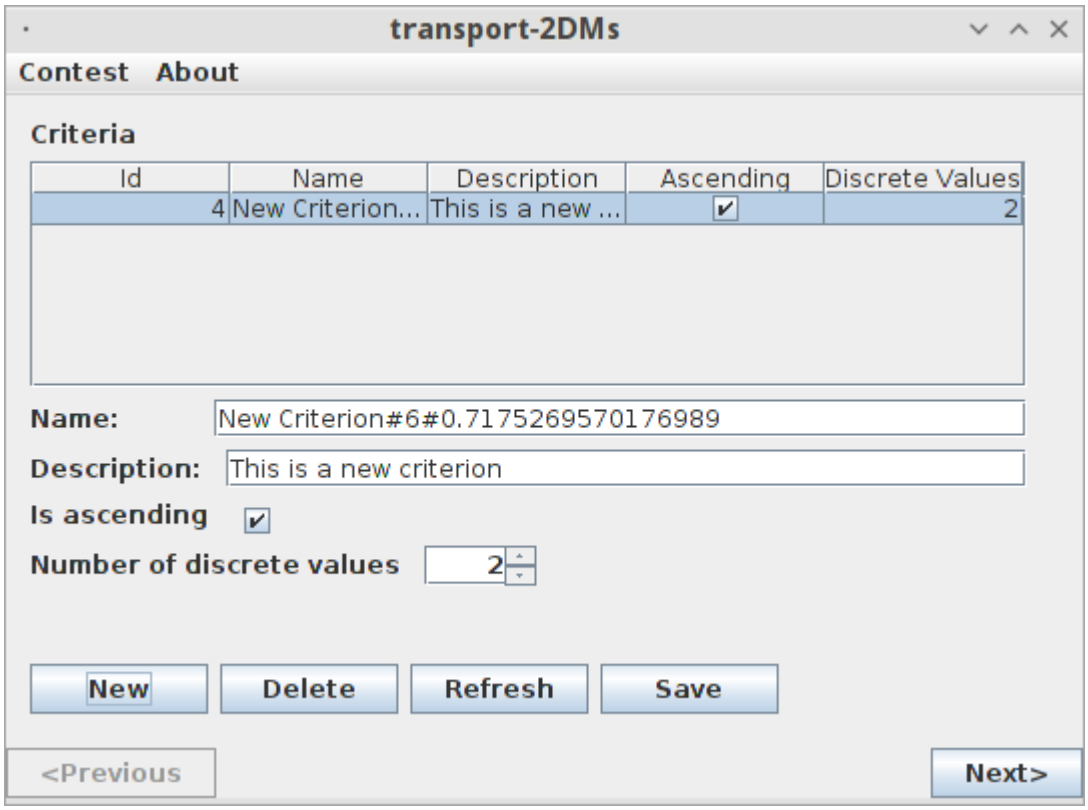


Figure 9: Entering a new criterion

By default, the application assigns an ID, an automatically-generated name and a generic "This is a new criterion" description" to the new criterion. It also checks, by default, the "ascending" option and considers the criterion to have two discrete values. Obviously, these will all need to be edited. The table of the criteria should look like this:

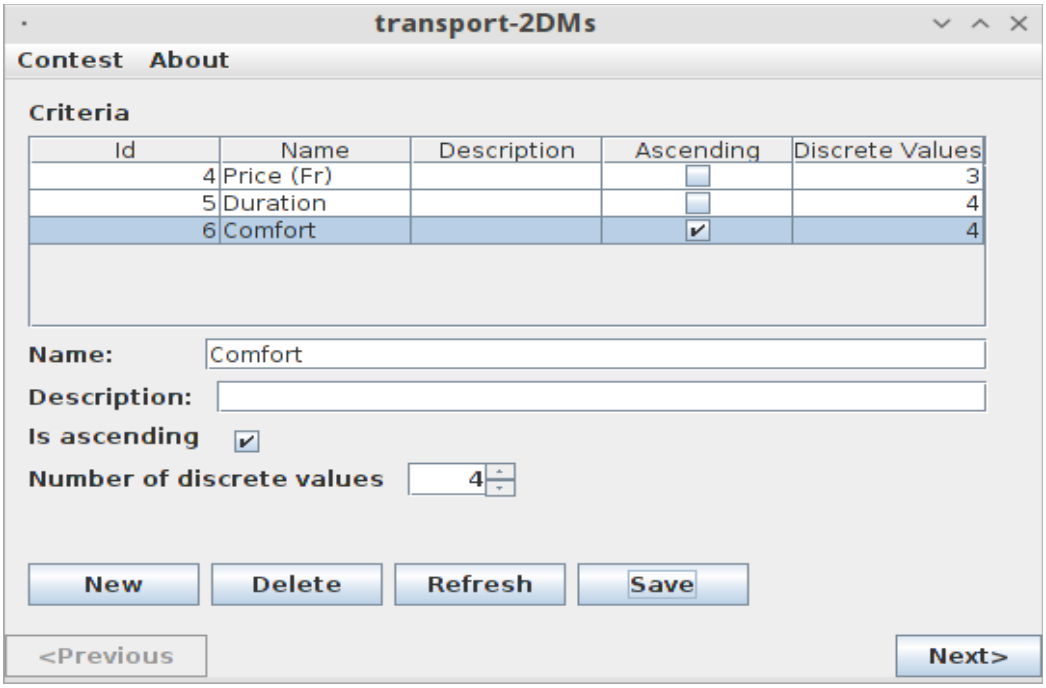


Figure 10. The criteria, after having been edited

Then, the user will have to click on "Next" to proceed to the next stage, where the number and names of DMs will be entered.

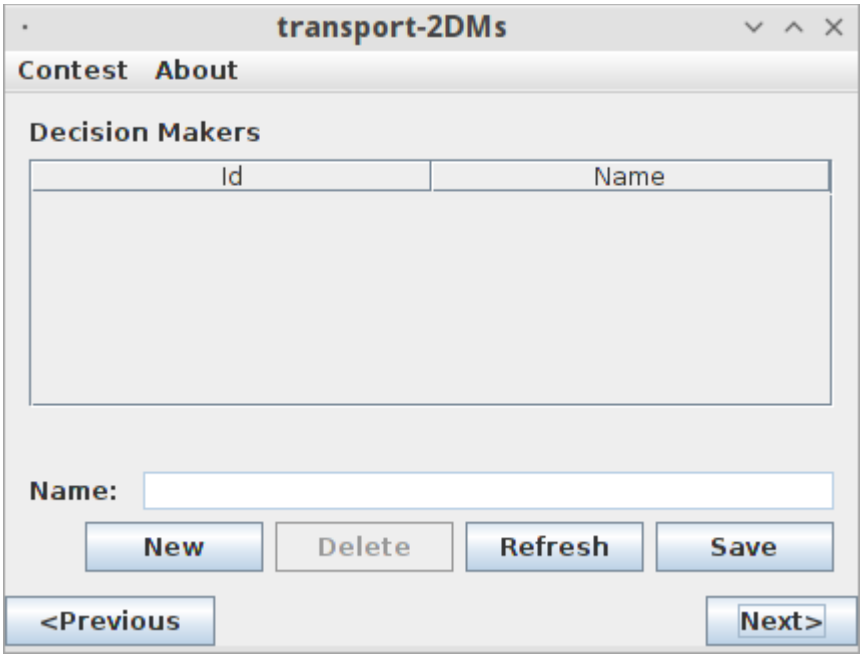


Figure 11. The DM entry window

Again, the procedure is similar. "New" allows the user to add a DM. "Delete" deletes a DM; "Refresh" ensures that the data are properly stored in case something went wrong.

"Save" stores the current state of the problem in the database. It must be noted that it is prudent to save after every change that is made to the data of the problem.

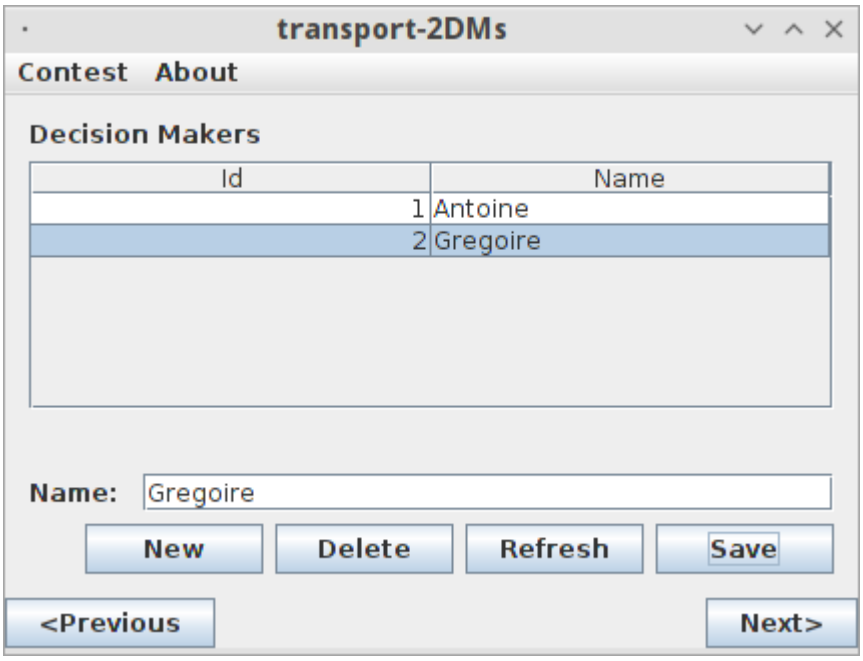


Figure 12. The DMs entered into the problem

Clicking "Next" takes the user to the Alternatives entry screen. Here, the user can enter the alternatives and their performances in each criterion.

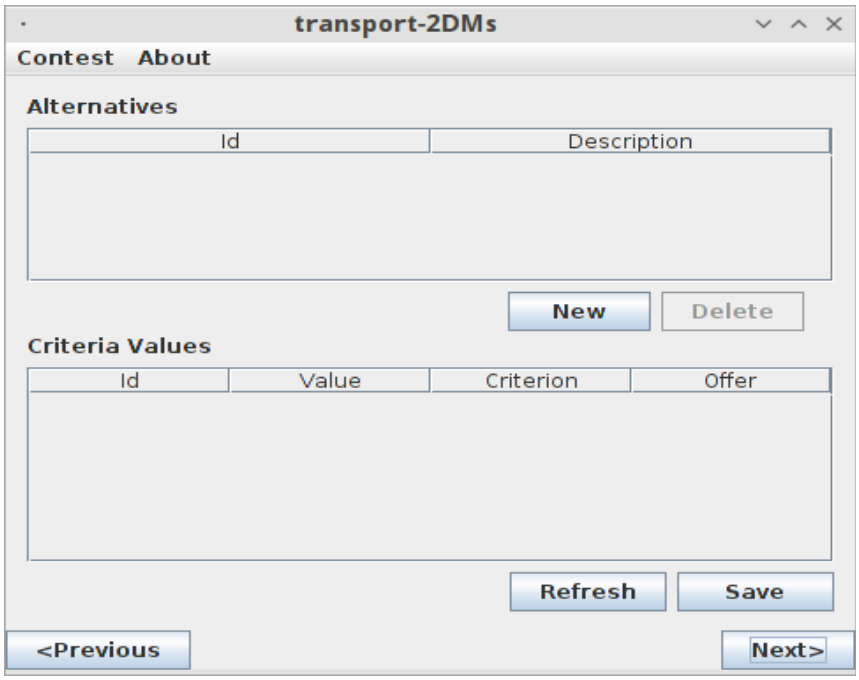


Figure 13. The Alternatives entry screen

Here, each alternative's description is its name. Also, the "value" in the window "Criteria Values" is the performance of the alternative in the respective criterion. Clicking "New" (for a new alternative) changes the window in the following manner:

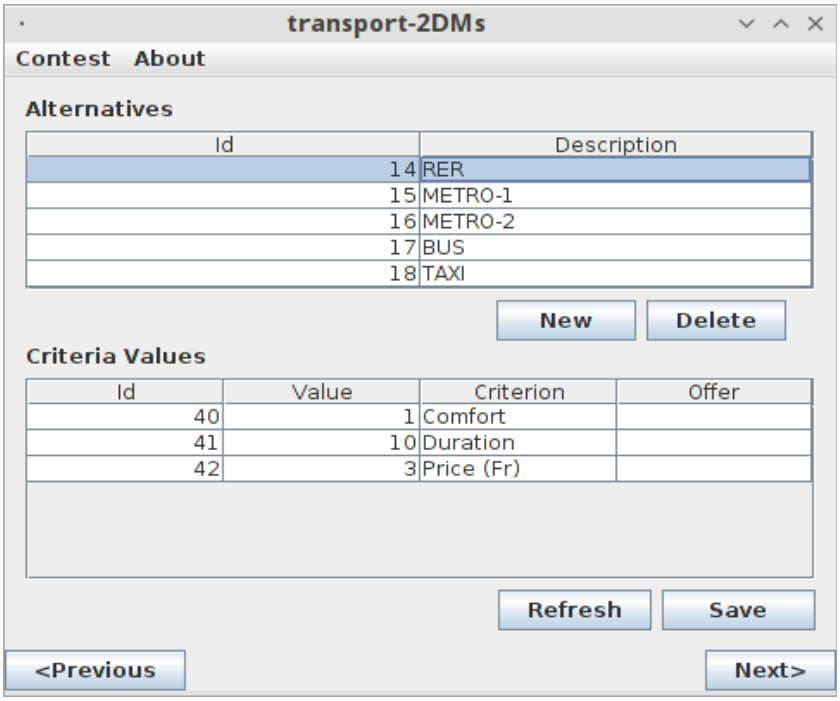


Figure 14. Entering alternatives and their performances.

The blue-highlighted field under "Description is where the name of the alternative will be entered, while the blue-highlighted field under "Value" is where the performance of the alternative will be entered. Note that the application arranges the criteria alphabetically. In the above screenshot, the performances of the alternative "RER" for the criteria of the decision problem are shown.

NOTE: The GUI of the application has an idiosyncracy similar to one exhibited by the EL-1S software that implements the ELECTRE I and ELECTRE IS methods: When the last value is entered in the "Value" (i.e. alternative performance) field, the user must highlight the adjacent cell in order for the value entered to register and remain also to register *only* for the alternative currently being edited; otherwise, this value will be applied to that cell in the next alternative as well.

After the user has entered the performances for all the alternatives of the problem, s/he must save the data entered so far and click "Next". If the application produces an error message prompting him/her to check the data s/he entered, this is merely an issue regarding the storage of the data in the databases and can be rectified by clicking "Previous" (thus going back to the screen of Figure ***12***) and then "Next" again, saving and clicking on "Refresh". In the next stage (Figure 15), the user is prompted to enter the weak ordinal rankings of the various DMs for the given alternatives. These are the Decision Makers' Rankings (DRs) of the UTASTAR method.

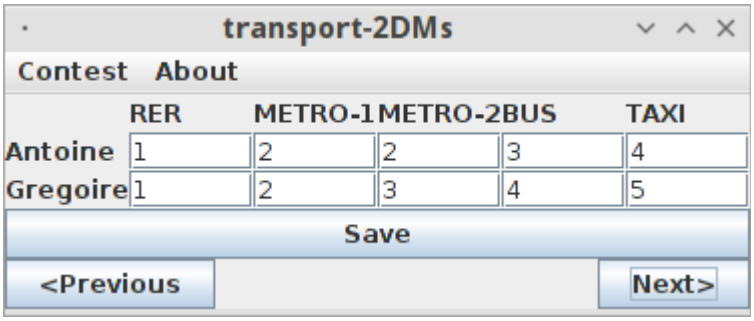


Figure 15. DR entry screen.

As is easily visible in the screenshot of Figure 15, the DR for the DM named "Antoine" is exactly the same as the one provided in the example of (Siskos and Yannacopoulos 1985). This is used to verify the results produced by the UTASTAR method. Figure 15 shows each alternative's marginal utility for each criterion for the DM named Antoine. Please note that the results for each DM are shown in diffenent tabs. The results for the DM named Gregoire are shown in Figure 17.

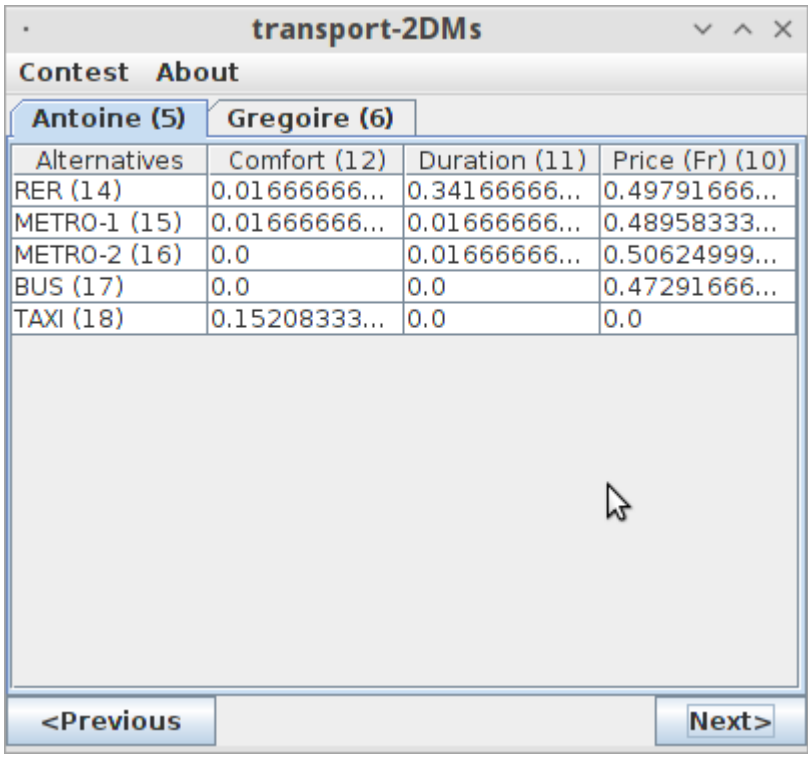


Figure 16. Marginal utility output screen – DM "Antoine"'s tab.

The numbers in brackets next to the names of the criteria, alternatives and DMs are the IDs automatically generated by the application and of course have no bearing to the results. Please note that the results are identical to the ones of the example presented by Siskos and Yannacopoulos (1985); this verifies that the UTASTAR method works properly.

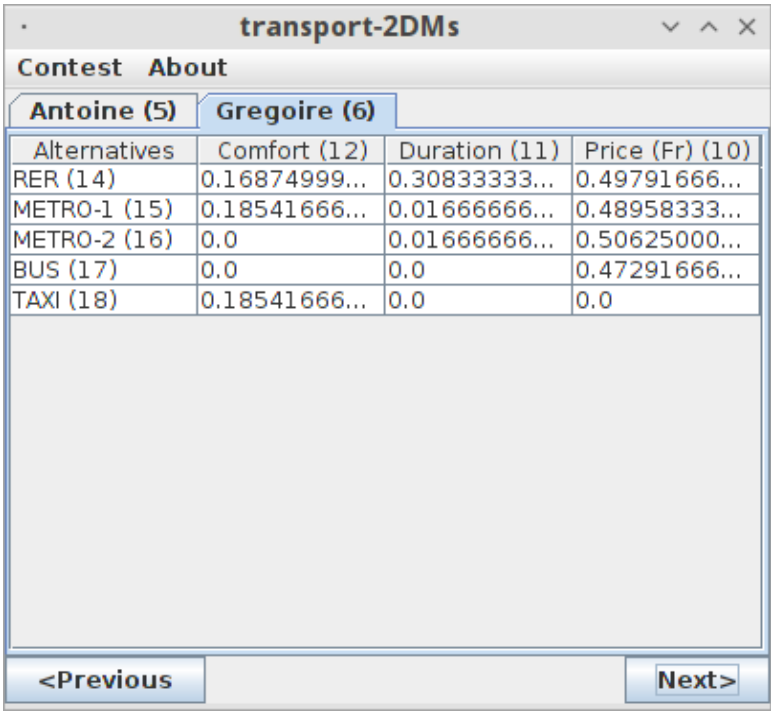


Figure 17. Marginal utility output screen – DM "Gregoire"'s tab.

Figure 18 shows the the global utilities for each alternative and for each DM, along with the MRs (Model Rankings) and the DRs (DR: Decision Maker's Ranking). In case an alternative's position in the MR is different from the DR, the difference is shown in red. If any DM wishes to change his/her DR or if the problem needs to be reformulated, they can move back to a previous step.

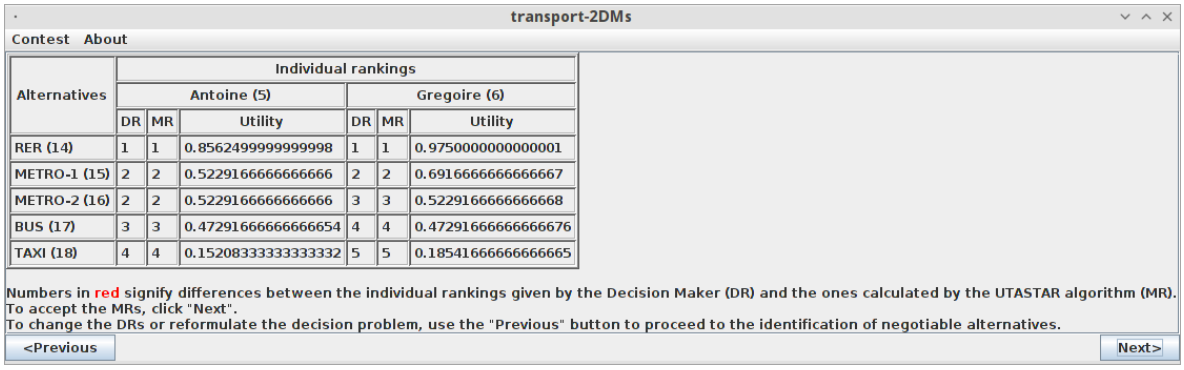
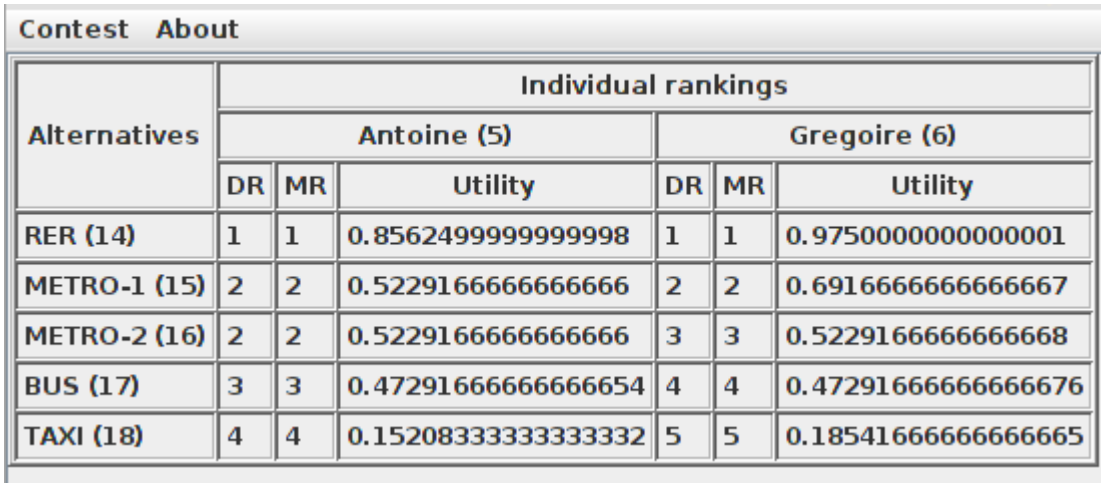


Figure 18. DR/MR output and global utility screen

To show the results in a larger size, the main table is shown in Figure 19.

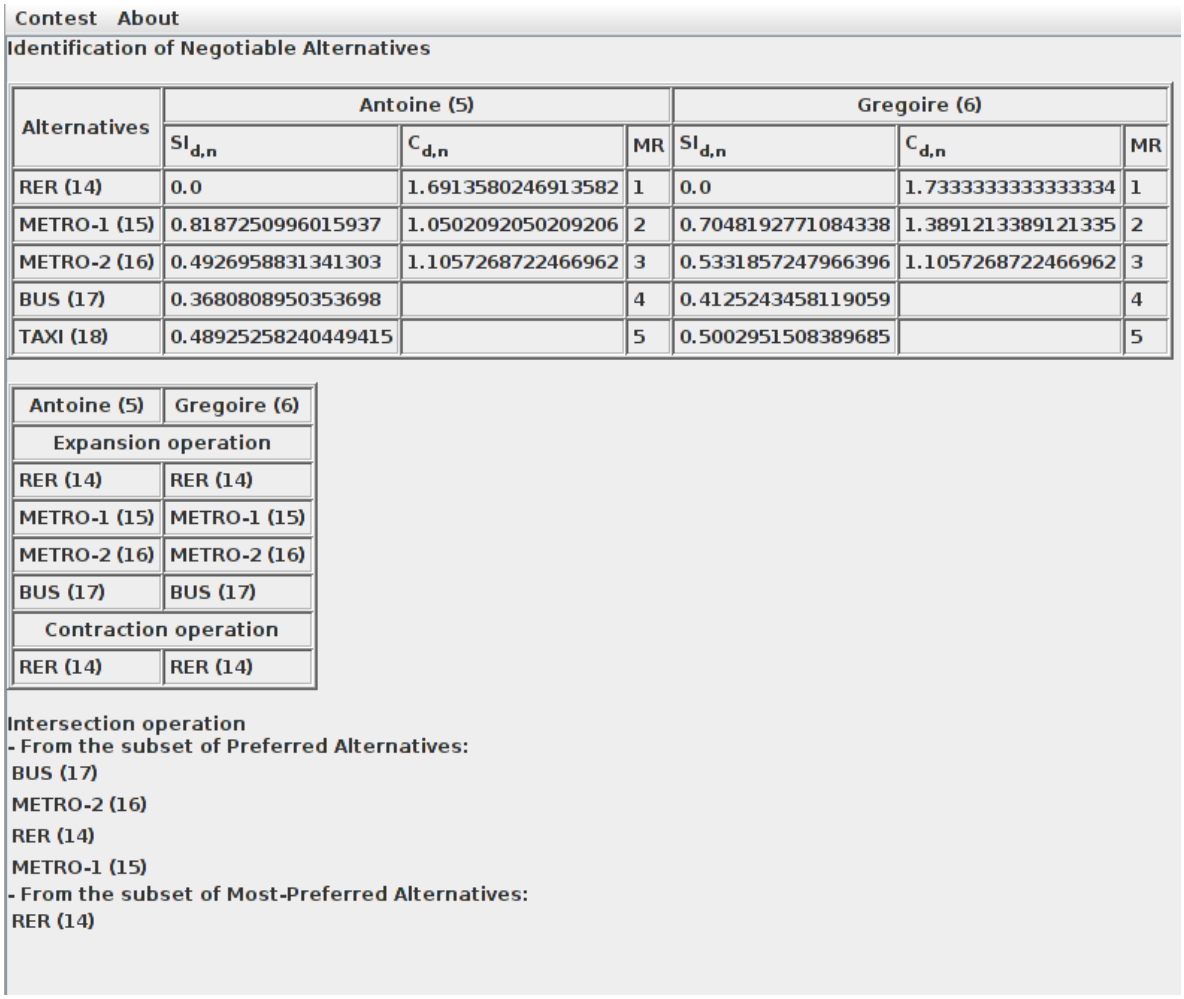


Contest About

Alternatives	Individual rankings					
	Antoine (5)			Gregoire (6)		
	DR	MR	Utility	DR	MR	Utility
RER (14)	1	1	0.8562499999999998	1	1	0.9750000000000001
METRO-1 (15)	2	2	0.5229166666666666	2	2	0.6916666666666667
METRO-2 (16)	2	2	0.5229166666666666	3	3	0.5229166666666668
BUS (17)	3	3	0.47291666666666654	4	4	0.47291666666666676
TAXI (18)	4	4	0.15208333333333332	5	5	0.18541666666666665

Figure 19. The results from Figure 17 (close-up)

The next step is to proceed to the NAI algorithm, which will produce the subsets of most preferred and preferred alternatives. It is noted again that the alternatives in the subset of preferred alternatives are the ones on which the agents in the *Argumentation Stage* will negotiate. The results are shown in Figure 20.



Contest About

Identification of Negotiable Alternatives

Alternatives	Antoine (5)			Gregoire (6)		
	$SI_{d,n}$	$C_{d,n}$	MR	$SI_{d,n}$	$C_{d,n}$	MR
RER (14)	0.0	1.6913580246913582	1	0.0	1.7333333333333334	1
METRO-1 (15)	0.8187250996015937	1.0502092050209206	2	0.7048192771084338	1.3891213389121335	2
METRO-2 (16)	0.4926958831341303	1.1057268722466962	3	0.5331857247966396	1.1057268722466962	3
BUS (17)	0.3680808950353698		4	0.4125243458119059		4
TAXI (18)	0.48925258240449415		5	0.5002951508389685		5

Antoine (5)	Gregoire (6)
Expansion operation	
RER (14)	RER (14)
METRO-1 (15)	METRO-1 (15)
METRO-2 (16)	METRO-2 (16)
BUS (17)	BUS (17)
Contraction operation	
RER (14)	RER (14)

Intersection operation
- From the subset of Preferred Alternatives:
BUS (17)
METRO-2 (16)
RER (14)
METRO-1 (15)
- From the subset of Most-Preferred Alternatives:
RER (14)

Figure 20. The results of the NAI algorithm

After the NAI algorithm comes the *Argumentation Stage*. In this stage, the protocol is run as many times as there are alternatives in the subset of preferred alternatives. It is apparent, however, that in certain problems (such as the one of this example), running the argumentation protocol may be redundant; for instance, in this example both DMs have the same alternative (RER) on top of their list, so they have basically agreed beforehand that this is the best solution for their problem. This is further cemented by the NAI algorithm, as the alternative RER is the only one in the subset of most preferred alternatives.

For this reason, in a future version of this application, a switch will be built into the system; it will alert the user(s) that their client-agents already agree to one alternative, therefore removing the need to proceed to the *Argumentation Stage*. So, when is the argumentation protocol necessary?

It is useful in the occasions where there are two or more alternatives in the subset of most preferred alternatives, as it will help the DMs' agents agree to *one* solution (after all, this methodology and its accompanying software solves a *choice* problem) and it will provide them with justification that is more easily comprehensible by human users. It is regrettable that an automatic text generator for the arguments could not be incorporated in

this software implementation. This is a feature that will be added in a future, updated and improved version.

The subset of preferred (therefore, negotiable) alternatives in this example consists of four alternatives, as the two DMs have essentially excluded the alternative "TAXI". The current implementation of the argumentation protocol will run the protocol four times, one for each alternative. The first of the four runs of the protocol is illustrated in Figure 21. When clicking "Next" on the NAI algorithm results screen, the user is taken to a blank screen, which has only the top menubar, the familiar "Previous" and "Next" buttons and the "Next Run" button. Clicking on the "Next Run" button makes the application perform the first run of the protocol.

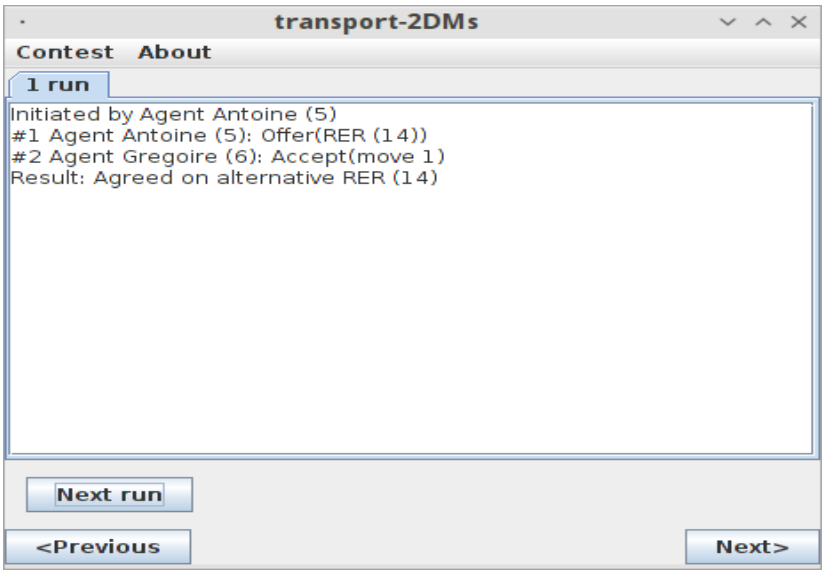


Figure 21. The first run of the argumentation protocol.

The run is initiated by the agent representing DM Antoine. The agent offers (proposes) RER. This alternative satisfies the goals of the agent representing DM Gregoire, so the agent accepts it. It is easily seen that the two agents (representing DMs Antoine and Gregoire) agree on this offer. In the next three runs, in which the agents take turns to make an offer, due to the aforementioned fact that both have the same alternative at the top of their respective lists, they make the same offer: RER.

III.4 Retrieving an existing problem from the database

Here, a problem that has already been saved in the software's database will be used. In this problem, four DMs are trying to choose *one* laptop from a set of fifteen. They have set seven criteria for their choice and the multicriteria table is as follows:

	CPU speed (GHz)	RAM (GB)	HD capacity (GB)	Price (EUR)	3D performance	2D performance	Weight (kg)
Laptop 1	2	4	500	450	6	7	1,8
Laptop 2	2,2	4	750	680	7	8	1,9
Laptop 3	2,2	8	750	890	8	9	3
Laptop 4	2,3	8	500	850	9	9	3
Laptop 5	1,8	8	750	400	4	6	1,7
Laptop 6	2	8	320	350	5	7	1,6
Laptop 7	1,8	2	320	300	3	5	1,6
Laptop 8	2,3	6	750	720	8	9	2,7
Laptop 9	2,3	8	1000	1180	10	10	4,6
Laptop 10	1,2	2	320	280	1	2	1,3
Laptop 11	2,3	8	750	800	9	10	2,9
Laptop 12	1,6	2	320	200	2	3	1,4
Laptop 13	1,2	2	250	220	1	1	1,1
Laptop 14	1,6	4	250	200	2	4	1,2
Laptop 15	2	8	320	500	8	8	2
Criteria scales	+	+	+	-	+	+	-
Discrete values	3	4	3	3	10	10	3

Table 4. The multicriteria table of the laptop choice problem.

The initial individual rankings of the alternatives by the DMs are as follows:

	DM1	DM2	DM3	DM4
Laptop 1	11	9	15	14
Laptop 2	9	11	1	13
Laptop 3	4	4	2	12
Laptop 4	8	8	6	10
Laptop 5	3	3	5	7
Laptop 6	15	15	7	6
Laptop 7	2	2	8	5
Laptop 8	1	1	11	1
Laptop 9	6	6	4	2
Laptop 10	5	5	3	15
Laptop 11	7	7	9	8
Laptop 12	14	14	14	11
Laptop 13	12	12	12	4
Laptop 14	10	10	13	3
Laptop 15	13	13	10	9

Table 5. The initial individual rankings of the alternatives by the DMs.

In the software that accompanies this thesis, a user can choose to retrieve (load) an already existing problem at any moment, stopping any editing currently being done to the problem that is open at that time. For illustrative purposes only however, it will be assumed that a new session of the software is initiated specifically for the problem that will be restored from the database.

So, the program is run as usual. This time, however, the user does not need to provide a name for the contest, as an already existing problem will be loaded. So, on the starting screen, s/he may click either on “OK” or on “Cancel”. This will take him to the next screen, where the criteria for the problem can be entered. Please note that, if no name is given to the problem, the program assigns a name-ID by itself. On the criteria entry screen, the user clicks on “Contest” and then “Load”. The program asks whether s/he wants to stop editing the current contest, to which the user's response is to click “Yes”.

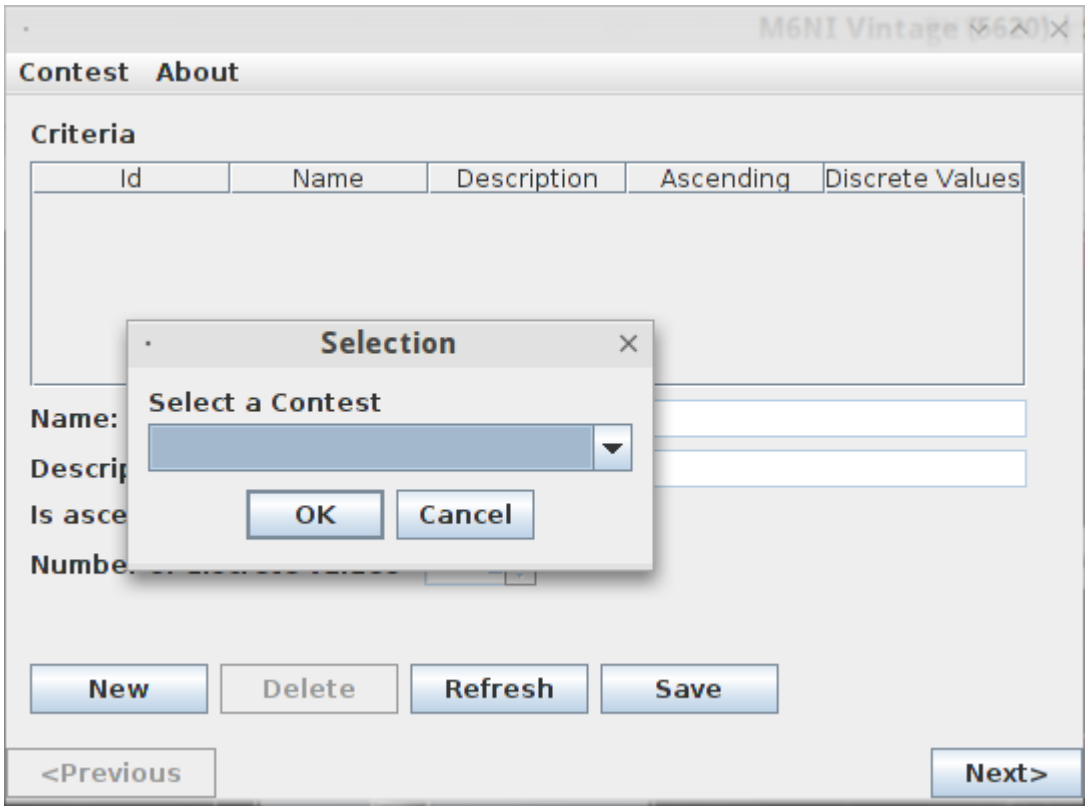


Figure 22. Contest selection.

From the pull-down menu, the user can choose the contest s/he wants by its name. In the case of this example, the name of the contest is “laptops”. Opening the contest presents the criteria that have already been set (Figure 23).

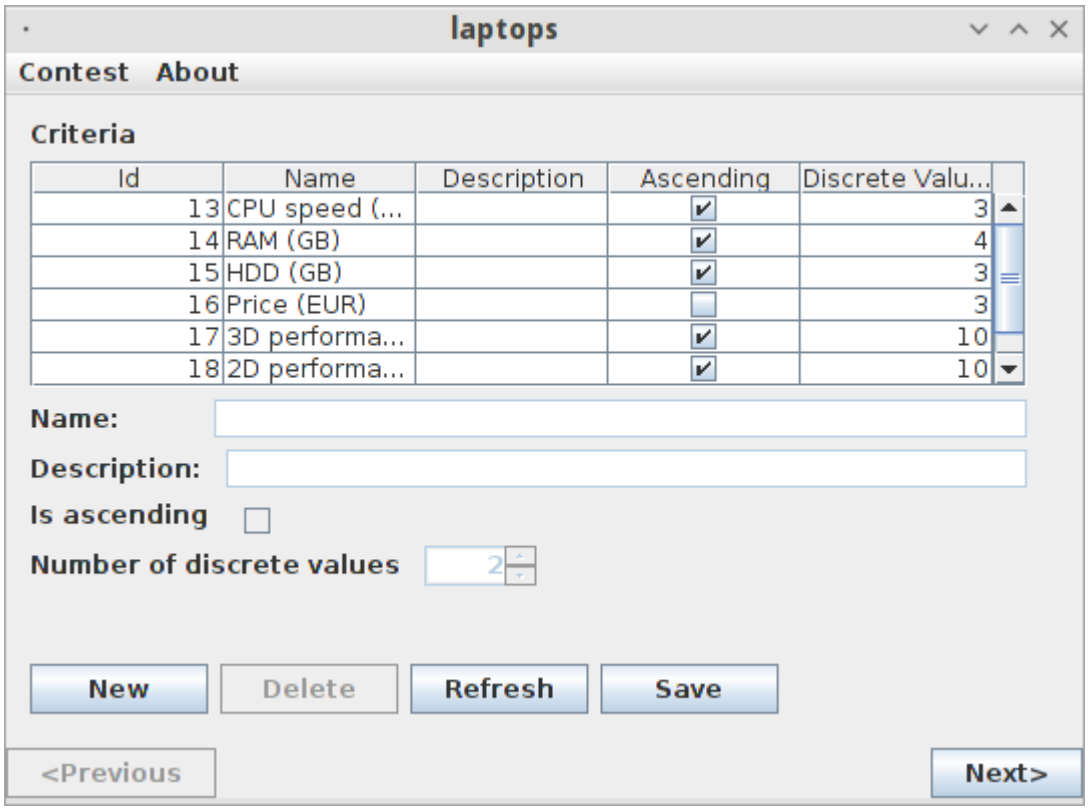


Figure 23. The criteria of the laptop choice problem.

Then, the system presents the list of DMs (Figure 24) that has already been provided in a previous session.

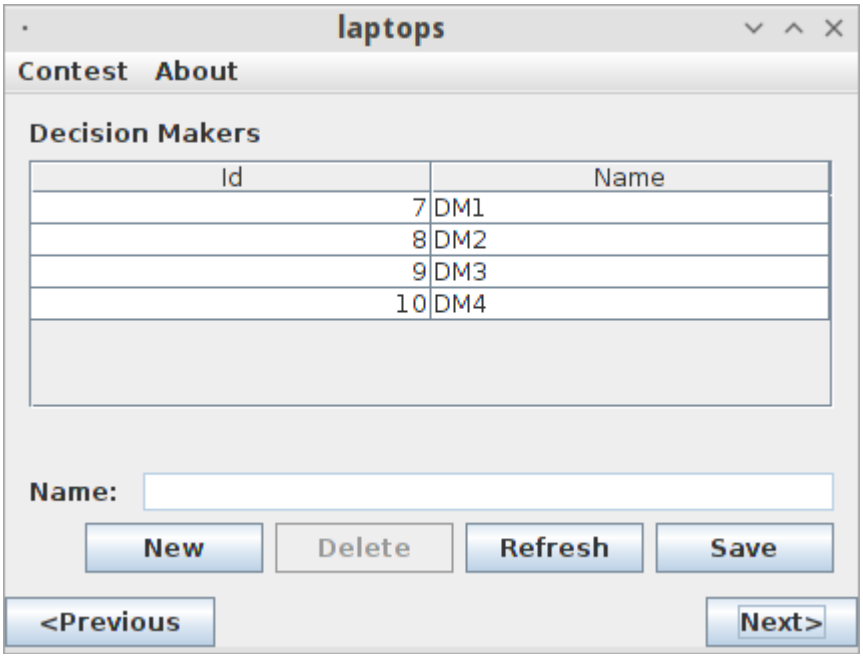


Figure 24. The list of DMs.

On the next screen (Figure 25), the list of alternatives is shown; in this particular screenshot, the alternative "Laptop 1" is highlighted, displaying its performances in the problem's criteria.

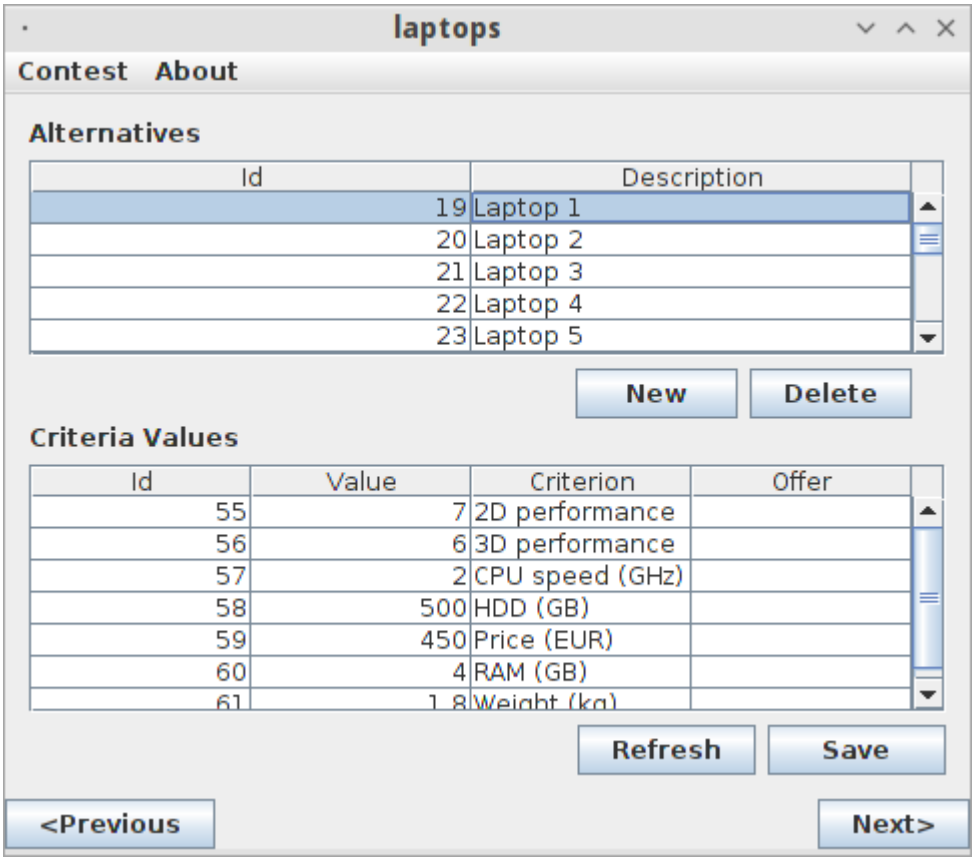


Figure 25. The list of alternatives.

As per the first example with the transport medium selection, what follows is the screen where the DMs' initial rankings were entered (Figure 26):

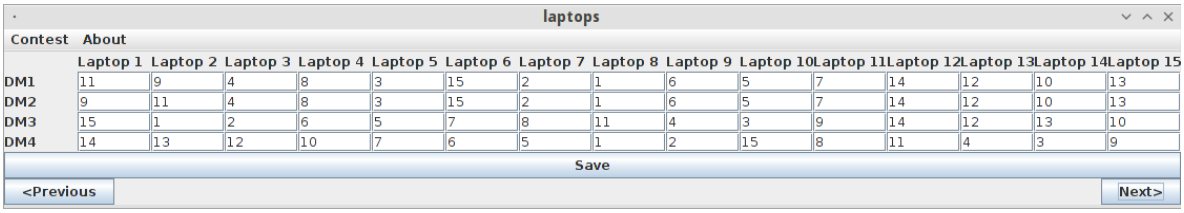


Figure 26. The DMs' initial rankings of the alternatives.

After the problem has been set, the UTASTAR method is applied for every DM. It calculates marginal and global utilities, as well as the weights of the criteria. It also points out differences between the MR and the DR for each DM and gives the DMs the option to either accept the changes it proposes for the rankings or to go back and reformulate the problem or modify their preferences (Figure 27).

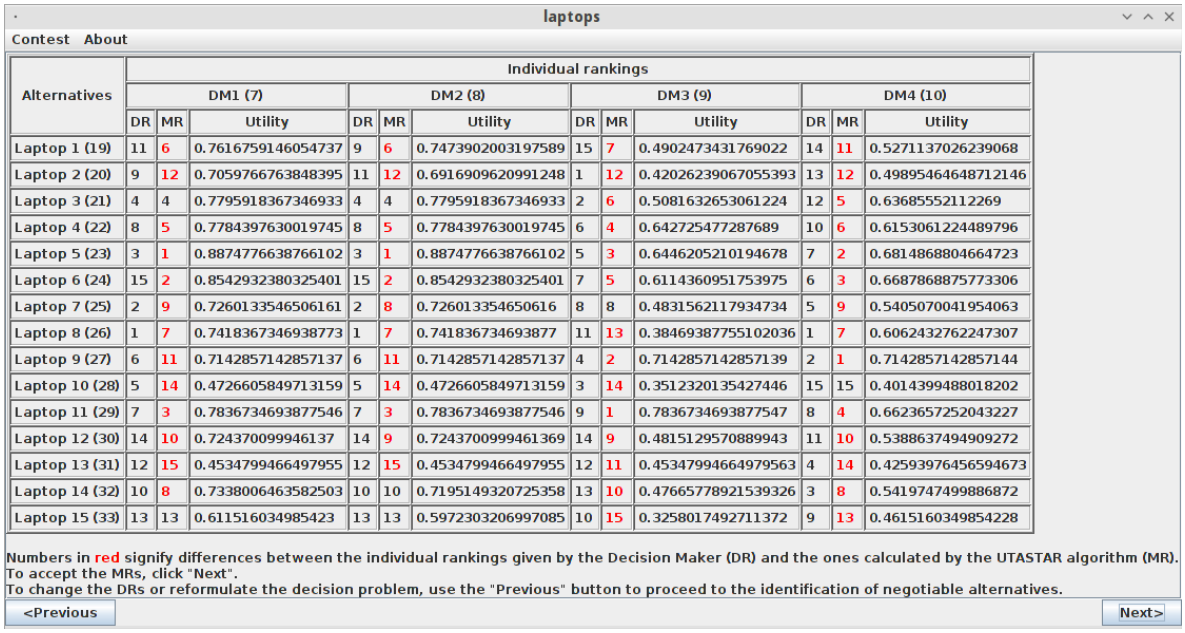


Figure 27. Global utilities and MRs as calculated by the UTASTAR method.

If the DMs accept these suggestions, the software applies the NAI algorithm to identify negotiable alternatives. The application of the NAI algorithm creates a subset of preferred alternatives that includes all the alternatives, but the subset of most preferred alternatives consists only of the alternative named “Laptop 11”.

After this step, the argumentation procedure commences. For brevity's sake, only the last run of the argumentation protocol is shown. As can be deduced from the results of the NAI algorithm, the DMs will again agree on the single solution that makes up the subset of most-preferred alternatives. It must also be noted that, since this particular problem has no alternatives that the DMs as a group would reject outright, none of the runs of the argumentation protocol ends with failure as a result of an agent's refusal to accept an offer.

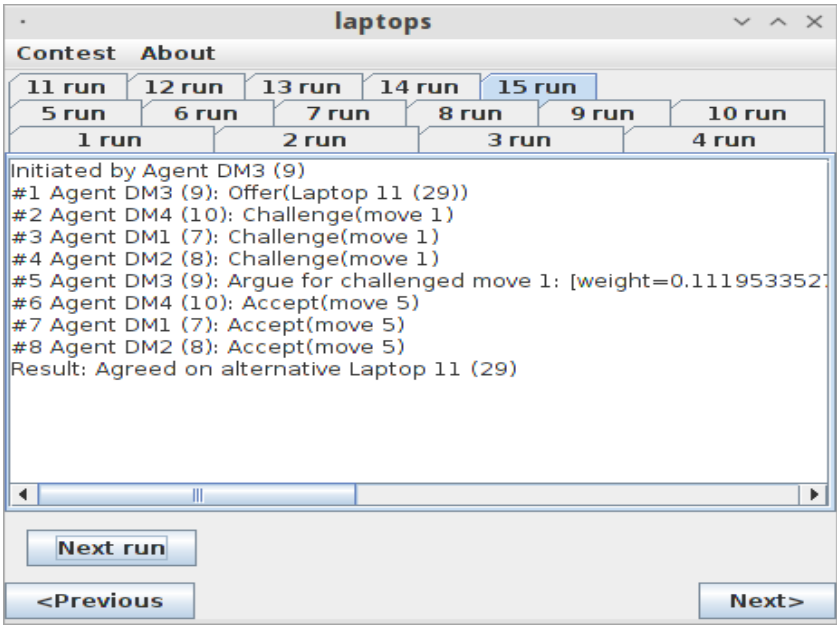


Figure 28. The final run of the argumentation protocol.

A third example, aimed at illustrating the system's ability to reduce the number of alternatives on which the agents will have to deliberate is defined in Table 6. It is a decision problem with ten alternatives, seven decision makers and four criteria.

	Criteria			
Alternatives	**1**	**2**	**3**	**4**
A1	500	1	600	3
A2	300	2	1000	4
A3	400	1	700	2
A4	300	4	600	5
A5	200	3	800	1
A6	400	3	1000	3
A7	500	5	500	5
A8	100	2	900	4
A9	400	4	600	2
A10	200	1	700	2
Criterion scale	+	-	+	-
Number of discrete values	5	5	5	5

Table 6. Multicriteria table.

	DMs and their individual rankings of the alternatives						
Alternatives	**DM1**	**DM2**	**DM3**	**DM4**	**DM5**	**DM6**	**DM7**
A1	1	4	2	6	6	7	1
A2	3	2	3	1	7	1	4
A3	4	3	6	8	2	5	2
A4	8	9	5	9	9	9	9
A5	5	6	8	4	1	4	7
A6	2	1	9	2	5	2	6
A7	9	10	10	10	10	10	10
A8	10	7	4	3	8	3	5
A9	7	8	7	7	4	8	8
A10	6	5	1	5	3	6	3

Table 7. The individual rankings of the alternatives according to the DMs.

As shown in Figure 29, the NAI algorithm here creates a subset of preferred alternatives consisting of seven alternatives instead of ten, while the subset of most-preferred alternatives consists of two alternatives:

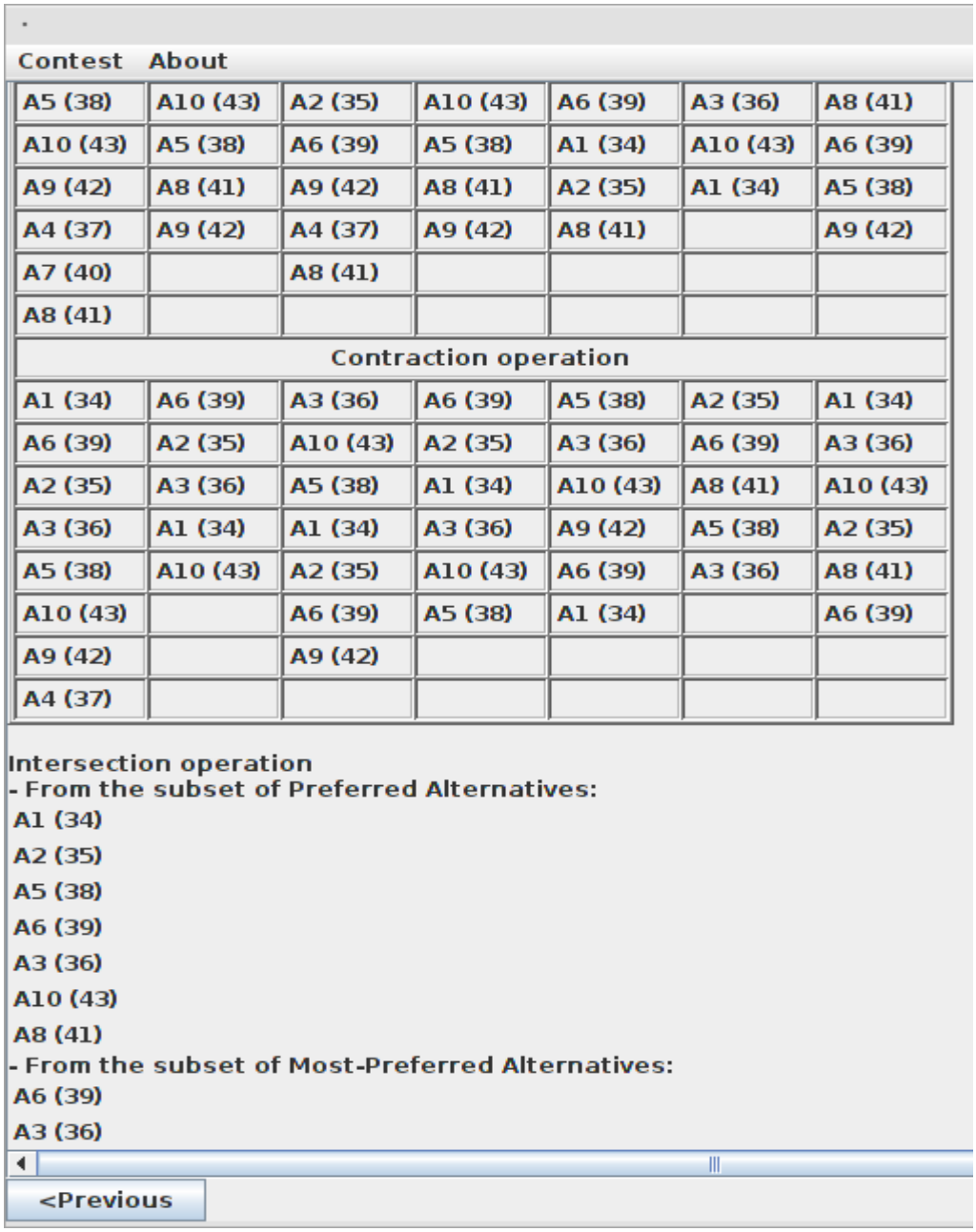


Figure 29. Application of the NAI algorithm in the aforementioned example.

In this example, the NAI algorithm creates a subset of negotiable alternatives that is considerably smaller than the original set. It must also be noted that the subset of most-preferred alternatives contains two alternatives, therefore even this subset, which in this methodological framework is not the one upon which the agents negotiate, can offer a basis for negotiation. Even so, it is clear that the system, after it runs the argumentation protocol, will recommend *one* of the alternatives in the most-preferred alternatives subset. Indeed, it finally recommends alternative A6, which is consistent with the placement of this alternative in the most-preferred alternatives subset.

III.5 Scope for Future Development

In spite of all the work that went into the development of the methodological framework and the accompanying software, the application cannot be considered complete. It is merely an alpha, perhaps even pre-alpha stage of an ambitious GDSS project. What was achieved with this application is that it was demonstrated that the use of a heuristic algorithm can, depending on the expressed preferences of the DMs (which are represented by intelligent, autonomous agents), accelerate the argumentation-based negotiation process by reducing the number of alternatives upon which the agents will negotiate. The NAI heuristic algorithm reduces the area in which the agents will search for a solution; from a large area that contains alternatives which the agents will not want to consider, to a smaller area, containing fewer alternatives that they will want to consider. The UTASTAR method is an excellent match for the NAI algorithm, as it provides the NAI algorithm with a suitable set of data (namely the global utilities, which express each DM's preference on an alternative, considering all the criteria) and provides both an ordinal and a cardinal ranking.

The argumentation protocol that was chosen to be adapted for this thesis was chosen because it provided the following benefits: (a) it can easily express its arguments in mathematical/numerical terms and, therefore, lends itself well to being implemented in programming languages that are not specifically made for Artificial Intelligence; it can be implemented in Java, C, C++, C# or any other such language, without dictating the need for a more specialized language like Prolog; (b) the fact that its arguments can be expressed in numerical terms means that it is easy to achieve interoperability with MCDA methods like those of the UTA family; (c) the fact that it is easy to implement in Java (in particular) makes it suitable for further development of this software, which will incorporate a distributed multi-agent system created with the current industry standard JADE/EJADE platform.

As has been mentioned above, though, the software is still in early stages of development and, although it produces the results predicted by the methodological framework on which it is based, there are many capabilities and enhancements that will be added in future versions, since its aim is to become a constantly evolving and useful GDSS for businesses:

- Transition from Netbeans to Eclipse, which is a more stable IDE and also has the advantage of the EJADE plug-in that aids the creation of multi-agent systems in Java.
- Improvements to the GUI, mostly for ergonomic reasons and conformity to universally accepted standards.
- Copy/paste capabilities will be added.
- Undo/Redo function will be added.
- Interoperability with spreadsheet applications (such as Microsoft's Excel, OpenOffice.org/LibreOffice's Calc, Gnumeric etc) needs to be implemented – at the very least, the software needs the ability to import and export data in .CSV format.
- Networking capabilities, along with session storage and retrieval, will be added in order to enable the presented system to become a true distributed GDSS.
- A web-based server-client architecture will be implemented, using a web browser as the interface for each user, with all the calculations taking place on the server side.

- For use in a corporate context, plug-ins will need to be developed such that it will easily exchange data with any existing corporate information system (such as an ERP system).
- Graph producing capabilities will be added.
- The argumentation stage will be enhanced with a text generator that will automatically generate arguments in a physical language easily comprehensible by human users.
- The system's engine itself will incorporate a number of useful enhancements that will enable it to become a more powerful tool for the DMs, namely:
    1. For qualitative criteria, the system will utilize different *values* in these criteria for *each* DM (in addition to the DMs' different marginal utilities that are based on common values in these criteria); that way, the fact that qualitative criteria express a subjective assessment of each alternative by each DM in a feature or characteristic that cannot be easily measured will be taken into account in a more complete manner.
    2. Different voting powers for each DM will be supported, thereby providing a better representation of situations where all DMs are not equal.
    3. More complex argument comparison principles will be implemented in order to more realistically capture the way humans assess and compare arguments provided to them by others during a negotiation.
    4. Conflict resolution capabilities will be added to the argumentation stage.

These issues are known to the author and the project will continue to be developed and improved over time until it reaches its full potential, with inclusion of at least another MCDA ranking method (more specifically, the UTAII and, perhaps, the Stochastic UTA, which is a version of the UTASTAR specially adapted to handle the probability of various scenarios occurring).

Chapter IV *Conclusions*

This thesis aims to develop a new methodological framework that combines the benefits of three techniques used in the (group) decision-making field: the UTASTAR multicriteria decision analysis method, which is used to enable the DMs to rank the alternatives at hand from best to worst, according to each one's preferential profile. The NAI heuristic algorithm, which is typically used to enable a group of DMs with different preferences to reach a compromise, a *consensus* on one commonly acceptable solution, but also enables them to narrow the scope of their negotiation, by removing the alternatives they reject outright and creating sets of negotiable alternatives that they would consider, even if they are not at the top of their lists. Finally, an argumentation-based negotiation protocol was adapted so that the final consensus would be reached and justification that is comprehensible by human users is provided. There is a significant departure from the typical approach of the protocol proposed by Amgoud, Belabbes and Prade (2005): whereas that initial protoco expresses arguments as functions of the alternative's *performances* in the criteria, this thesis expresses arguments as functions of the *utilities*, enhancing and strengthening the argumentation protocol's ability to take into account the subjective preferences of the DMs.

The methodological framework proposed in this thesis has been implemented in a software application written entirely in the Java programming language, which has been chosen for its cross-platform nature that will enable this system to be used on every current important operating system: Windows, GNU/Linux, Mac OS X, Android, iOS. The system presented here provides an intuitive, easy-to-use Graphical User Interface (GUI) that, in the same vein as wizard-style applications, guides the user from one step to the next; it also allows the user to go back to a previous step in order to correct or modify data s/he has entered.

It has been demonstrated through examples in this thesis that the combination of heuristics and argumentation can, under certain conditions, accelerate the group decision-making process. This is not always the case, however: the overall performance of the system depends on the complexity of the DMs' preferences.

IV.1 Future work

In the future, this system will be further developed, with refinements and improvements that will be incorporated in its methodological framework (the theoretical side) and enhancements that will be added to its functionality (the software side). On the theoretical

side, reference set selection and extrapolation capabilities will be added to the UTASTAR algorithm and at least a second MCDA method will become available to the user (the UTAII and perhaps the Stochastic UTA), along with improved modeling of DMs' subjective assessment of alternatives; to achieve this, a future version of the system presented in this thesis will support different *values* for each DM in qualitative criteria, as well as each DM's already supported marginal utilities for these criteria. Also, regarding the argumentation stage, in addition to the implementation of more complex and advanced argument comparison principles (which will more realistically capture the manner in which human DMs assess arguments presented to them in a negotiation setting), the merit of a stratification of the alternatives in the *argumentation stage* (in addition to their stratification regarding their desirability by the NAI heuristic algorithm) will be explored regarding its plausibility and usefulness. The argumentation stage will also be enhanced with conflict resolution capabilities, part of which will be the aforementioned stratification of the alternatives. Furthermore, the option of having the agents negotiate on the subset of most-preferred alternatives rather than the subset of preferred alternatives will be explored. This will mean that resorting to the argumentation stage might not always be necessary; instead, the argumentation stage could become a step for the achievement of consensus in the cases where the NAI algorithm fails (for instance, when there are more than one alternatives in the subset of most-preferred alternatives).

On the software side, the accompanying application will be further developed so that this system will become a mature, complete distributed multi-agent GDSS capable of being used on a multitude of platforms (from Android to Windows and from Mac OS X to GNU/Linux). Enhancements that will be added in the future will include:

- Transition from Netbeans to Eclipse, which is a more stable IDE and also has the advantage of the EJADE plug-in that aids the creation of multi-agent systems in Java.
- Improvements to the GUI, mostly for ergonomic reasons and conformity to universally accepted standards.
- Copy/paste capabilities will be added.
- Undo/Redo function will be added.
- Interoperability with spreadsheet applications (such as Microsoft's Excel, OpenOffice.org/LibreOffice's Calc, Gnumeric etc) needs to be implemented – at the very least, the software needs the ability to import and export data in .CSV format.
- Networking capabilities, along with session storage and retrieval, will be added in order to enable the presented system to become a true distributed GDSS.
- A web-based server-client architecture will be implemented, using a web browser as the interface for each user, with all the calculations taking place on the server side.
- For use in a corporate context, plug-ins will need to be developed such that it will easily exchange data with any existing corporate information system (such as an ERP system).
- Graph producing capabilities will be added.
- The argumentation stage will be enhanced with a text generator that will automatically generate arguments in a physical language easily comprehensible by human users.
- The system's engine itself will incorporate a number of useful enhancements that will enable it to become a more powerful tool for the DMs, namely:

1. For qualitative criteria, the system will utilize different *values* in these criteria for *each* DM (in addition to the DMs' different marginal utilities that are based on common values in these criteria); that way, the fact that qualitative criteria express a subjective assessment of each alternative by each DM in a feature or characteristic that cannot be easily measured will be taken into account in a more complete manner.
2. Different voting powers for each DM will be supported, thereby providing a better representation of situations where all DMs are not equal.
3. More complex argument comparison principles will be implemented in order to more realistically capture the way humans assess and compare arguments provided to them by others during a negotiation.
4. Conflict resolution capabilities will be added to the argumentation stage.

The work done on this thesis will also signify the beginning of a concerted effort to develop a set of reusable, modular software libraries implementing MCDA methods, consensus-seeking algorithms and argumentation protocols; all these libraries will be written in Java and, if resources permit, in C and/or C++; this will enable future researchers and developers to create new DSS and GDSS systems easier, faster and more effectively.

REFERENCES

[1] Amgoud, L., Bonnefon J-F., Prade, H. 2005, "An argumentation-based approach to multiple criteria decision". In *Proceedings of the European Conference on Symbolic and Quantitative Approaches to Reasoning with Uncertainty (ECSQARU'05)*, pp. 269-280.

[2] Amgoud L., Prade H., Belabbes S. 2005, "Towards a formal framework for the search of a consensus between autonomous agents". In *AAMAS'05*

[3] Arús, C. et al. 2005, "On the design of a web-based decision support system for brain tumour diagnosis using distributed agents". In *Int. Conf. on Intelligent Agent Technology* (Hong Kong, Dec. 2006), IEEE.

[4] Barzilai, J. and F. A. Lootsma. 1997, "Power Relations and Group Aggregation in the Multiplicative AHP and SMART", *Journal of Multi-Criteria Decision Analysis* 6, pp. 155–165.

[5] Boella, G., Hulstijn, J. and van der Torre, L. 2006, "A Logic of Abstract Argumentation", in *Postproceedings of ArgMAS 2005*, LNAI 4049, pp.29-41, Springer-Verlag Berlin Heidelberg.

[6] Bogetoft, P. and P. Pruzan. 1991, *Planning with Multiple Criteria: Investigation, Communication, Choice*. Amsterdam, North Holland.

[7] Bonnefon, J. F., Glasspool, D., McCloy, R., and Yule, P. 2005, *Qualitative decision making: Competing methods for the aggregation of arguments*. Technical report, 2005.

[8] Bui, T. X. 1987, *Co-oP: A Group Decision Support System for Cooperative Multiple Criteria Group Decision Making*, Berlin, Springer-Verlag.

[9] Bui, T. X. and M. Jarke. 1986, "Communications Design for Co-oP: A Group Decision Support System", *ACM Transactions on Office Information Systems* **4,** 2.

[10] Bui, T. X. and Yen, J. 1995, "The Negotiable Alternatives Identifier (NAI) for Negotiation Support – An Improved Algorithm", *Proceedings of the Third International Conference on Decision Support Systems*, Hong Kong, June 22-23, 1995, Hong Kong University of Science and Technology, Hong Kong, 1995, pp. 149-159

[11] Carlsson, C., Ehrenberg, D., Eklund, P., Fedrizzi, M., Gustafsson, P., Lindholm, P., Merkuryeva, G., Riisanen, T., and Ventre, A.G.S. 1992, "Consensus in Distributed Soft Environments," *European Journal of Operational Research* **61**, pp. 165–185.

[12] Choi, H.-A., Suh, E.-H. and Suh, C.-K. ,1994, "Analytic Hierarchy Process: It Can Work for Group Decision Support Systems", *Computers and Industrial Engineering* **27**(1–4), pp. 167–171.
[13] Colson, G. and Mareschal, B. 1994, "JUDGES: A Descriptive Group Decision Support System for the Ranking of the Items", *Decision Support Systems* **12**, pp. 391–404.
[14] Csáki, P., Csiszár, L., Fölsz, F., Keller, K., Mészáros, Cs., Rapcsák, T. and Turchányi, P. 1995a, "A Flexible Framework for Group Decision Support, WINGDSS Version 3.0", *Annals of Operations Research* 58, pp. 441–453.
[15] Csáki, P., Rapcsák, T., Turchányi, P., and Vermes, M. 1995b, "R and D for Group Decision aid in Hungary by WINGDSS, A Microsoft Windows Based Group Decision Support System", *Decision Support Systems* **14**, pp. 205–217.
[16] Dimopoulos, Y., Moraïtis, P., Tsoukiàs, A. 2003, "Argumentation based modelling of decision aiding for autonomous agents". In *Proceedings of the IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT'04)*.
[17] Dose, J. J. 2003, "Information Exchange in Personnel Selection Decisions", *Applied Psychology*: *An International Review* 52(2), 237–252.
[18] Dung, P.M. 1995, "On the acceptability of arguments and its fundamental role in non-monotonic reasoning, logic programming and *n*-person games". *Artificial Intelligence* **77**, pp. 321-357.
[19] Dyer, R. F. and Forman, E. H. 1992, "Group Decision Support with the Analytic Hierarchy Process," *Decision Support Systems* 8, pp. 99–124.
[20] Felsenthal, D. S. and Machover, M. 2004, "A Priori Voting Power: What Is It All About?" *Political Studies Review* 2, pp. 1–23.
[21] Finlay, P.N. 1994, *Introducing decision support systems*. Oxford, UK Cambridge, Mass., NCC Blackwell; Blackwell Publishers.
[22] Fox, J., Krause, P. and Ambler, S. 1992, "Arguments, contradictions and practical reasoning". In *Proceedings of the 10th European Conference on Artificial Intelligence (ECAI-92), Vienna, Austria,* pp. 623-627.
[23] González-Vélez, H. et al. 2006, "Agent-based distributed decision support system for brain tumour diagnosis and prognosis". In *Current Research in Information Sciences and Technologies. Multidisciplinary approaches to global information systems* (Merida, Spain, Oct. 2006), University of Extremadura and the Open Institute of Knowledge.
[24] Giordano, J. L., Jacquet-Lagrèze E., and Shakun, M. F. 1988, "A Decision Support System for Design and Negotiation of New Products," in Shakun, M. F. (Ed.), *Evolutionary Systems Design*. Oakland, CA, Holden-Day.
[25] Guttman, R.H. and Maes, P. 1998, Agent-mediated integrative negotiation for retail electronic commerce. In *Proceedings of the Workshop on Agent Mediated Electronic Trading (AMET'98)*, pp. 70-90.
[26] Hollingshead, A. B. 1996, "The Rank-Order Effect in Group Decision Making," *Organizational Behavior and Human Decision Processes* 68(3), 181–193.
[27] Hwang, C. L. and Ling M. J. 1987, *Group Decision Making Under Multiple Criteria*. Lecture Notes in Economics and Mathematical Systems, Vol. 281. Berlin, Springer-Verlag.
[28] Ito, T. and Toramatsu, S. 1997, "Persuasion among agents: an approach to implementing a group decision support system based on multi-agent

negotiation". *In Proceedings of the 5th International Joint Conference on Artificial Intelligence (IJCAI'97)*, pp. 592-597.

[29] Iz, P. and L. Krajewski. 1992, "Comparative evaluation of three interactive multiobjective programming techniques as group decision support tools", *INFOR*. 30(4), pp. 349–363.

[30] Jacquet-Lagrèze, E. and Siskos, Y. 1982, "Assessing a Set of Additive Utility Functions for Multicriteria Decision Making: The UTA Method", *European Journal of Operational Research* 10(2), pp. 151–164.

[31] Jacquet-Lagrèze, E. and Siskos, Y. 2001, "Preference Disaggregation: 20 Years of MCDA Experience", *European Journal of Operational Research* **130**(2), 233–245.

[32] Jarke, M. 1986, "Knowledge Sharing and Negotiation Support in Multiperson Decision Support Systems," *Decision Support Systems* **2**, 93–102.

[33] Jarke, M., M.T. Jelassi, and Shakun. M. F. 1987, "MEDIATOR:Toward aNegotiation Support System," *European Journal of Operational Research* 31(3), 314–334.

[34] Jelassi, M. T., Kersten, G., and Zionts, G. 1990, "An introduction to Group Decision and Negotiation Support," In Bana e Costa, C. A. (Ed.), *Readings in Multiple Criteria Decision Aid*. Berlin, Springer-Verlag.

[35] Jennings, N.R. *et al* 2001, Automated negotiation: prospects, methods and challenges. *International Journal of Group Decision and Negotiation,* **10**(2), pp. 199-215.

[36] Jowett, B. 1875, *The Dialogues of Plato*, 2nd edition. Oxford University Press, Oxford.

[37] Kakas, A. and Moraïtis, P. 2006, Adaptive agent negotiation via argumentation. In *Proc AAMAS'06.*

[38] Kakas, A. and Moraïtis, P. 2003, Argumentation-basd decision making for autonomous agents. In *Proceedings of the 2nd International Joint Conference on Autonomous Agents and multiagent systems*, pages 883-890, Melbourne.

[39] M. Kalech and Pfeffer, A. 2010, "Decision making with dynamically arriving information". In *Proc. AAMAS 2010*, pp. 267-274.

[40] Karacapilidis, N. and Papadias, D. 1997, "A group decision and negotiation support system for argumentation based reasoning". In Gr. Antoniou & M. Truzczynski (eds.) *Learning and Reasoning with Complex Representations*, Lecture Notes in AI, Springer-Verlag, Berlin 1997.

[41] Karacapilidis, N. and Trousse, B. 1997, "Computer-supported argumentation for cooperative design on the world-wide web". In *Proceedings of the 2 nd International Workshop on CSCW in Design (CSCWD'97)*, pp. 96-103.

[42] Karacapilidis, N. and Papadias, D. 1998, "Hermes: supporting argumentative discourse in multi-agent decision making", In Proceedings of the *American Association for Artificial Intelligence,* 1998.

[43] Karacapilidis, N. and Moraïtis, P. 2001, "Building an agent-mediated electronic commerce system with decision analysis features". In *Decision Support Systems* 32(2001), pp. 53-69.

[44] Keen, P.G.W. And Scott-Morton M. S. 1978, *Decision support systems: an organizational perspective*. Reading, Mass., Addison-Wesley Pub. Co.

[45] Keeney, R. L. 1992, *Value Focused Thinking. A Path to Creative Decision Making*. Cambridge, MA, Harvard University Press.

[46] Keeney, R. L. and Raiffa, H. 1976, *Decisions with multiple objectives*: *Preferences and value tradeoffs*. New York, Wiley.
[47] Kersten, G. E. 1985, "NEGO-Group Decision Support System," *Information and Management* 8(5), pp. 237–246.
[48] Kersten, G. E. and Szapiro, T. 1985, "Generalized Approach to Modelling Negotiations," *European Journal of Operational Research* 26, pp. 124–142.
[49] Kersten, G. E. 1987, "On Two Roles Decision Support Systems Can Play in Negotiations," *Information Processing and Management* 23(6), pp. 605–614.
[50] Kirkwood, C. W. 1997, *Strategic Decision Making*: *Multiobjective Decision Analysis With Spreadsheets*. Belmont, CA, Duxbury Press.
[51] Kraus, S., Sycara, K., and Evenchik A. 1998, "Reaching agreements through argumentation: a logical model and implementation". In *Artificial Intelligence,* vol. 104, pp. 1-69.
[52] Laruelle, A. and Widgren, M. 2000, *Voting Power in a Sequence of Cooperative Games: The Case of EU Procedures, Homo Oeconomicus XVII*, 67-84. Reprint in Holler M. J. and G. Owen (Eds.), 2001, *Power Indices and Coalition Formation*, Kluwer Academic Publishers, pp. 253–271.
[53] Leech, D. 2002, *Computation of Power Indices*, Warwick Economic Research Papers, No. 644.
[54] Lewandowski, A. 1989, "SCDAS-Decision Support System for Group Decision Making: Decision Theoretic Framework," *Decision Support Systems* **5**.
[55] Lluch-Ariet M. et al. 2008, "HealthAgents: Agent-Based Distributed Decision Support System for Brain Tumour Diagnosis and Prognosis". In *Agent Technology and e-Health*. Basel: Birkauser pp. 5-24.
[56] Lootsma, F. A. 1993, "Scale Sensitivity in the Multiplicative AHP and SMART," *Journal of Multi-Criteria Decision Analysis* 2, pp. 87–110.
[57] Maes, P., Guttman, R.H. and Moukas, A. 1999, "Agents that buy and sell". In *Communications of the ACM,* March 1999/vol.42, no.3*,* pp. 81-91.
[58] McBurney, P., van Eijk, R., Parsons, S., Amgoud, L. (2001) "A dialogue-game protocol for agent purchase negotiations". *Journal of Autonomous Agents and Multi-Agent Systems*, **7** (3). pp. 235-273.
[59] Matsatsinis, N. F., and Siskos, Y. 1999, "MARKEX: An intelligent decision support system for product development decisions," *European Journal of Operational Research,* **113** (2), pp. 336-354.
[60] Matsatsinis, N. F., Moraitis, P., Psomatakis, V., Spanoudakis, N. 1999, "Intelligent software agents for products penetration strategy selection". In proccedings of MAAMAW'99 (CD, *Modelling Autonomous Agents in a Multi-Agent World,* Valencia 30/6 – 2/7, Spain, 1999).
[61] Matsatsinis, N. F. and Siskos, Y. 1999, "MARKEX: An Intelligent Decision Support System for Product Development Decisions," *European Journal of Operational Research* **113**(2), pp. 336–354.
[62] Matsatsinis, N. F. and Samaras, A. P., 2000, "Brand Choice Model Selection Based on Consumers' Multicriteria Preferences and Experts' Knowledge". *Computers and Operations Research* 27(7/8), pp. 689–707.
[63] Matsatsinis, N. F. and Samaras, A. P. 2001, "MCDA and Preference Disaggregation in Group Decision Support Systems," *European Journal of Operational Research* 130(2), pp. 414–429.

[64] Matsatsinis, N.F. and Siskos, Y. 2002, *Intelligent support systems for marketing decisions*, Kluwer Academic Publishers.

[65] Matsatsinis, N.F., Moraïtis, P., Psomatakis, V., Spanoudakis N. 2003, "An Agent-Based System for Products Penetration Strategy Selection", *Applied Artificial Intelligence: An International Journal*, vol. 17, no. 10, pp. 901-925.

[66] Moraïtis, P. and Tsoukiàs, A. 2003, "Decision aiding and argumentation". In *Proc. of the 1st European Workshop on Multi-Agent Systems*.

[67] Nachenberg, C. S. 2002, *Dynamic heuristic method for detecting computer viruses using decryption exploration and evaluation phases* [Online – US Patent 6,357,008 (application year: 1997)] Available: http://service1.symantec.com/legal/publishedpatents.nsf/0/4b4a30633137923b88256df7005d6b5d/$FILE/United%20States%20Patent%206,357,008.htm [Accessed 23 Mar. 2012].

[68] Noori, H. 1995, The Design of an Integrated Group Decision Support System for Technology Assessment, *R& D Management* **25**(3), pp. 309-322.

[69] Parsons, S. and Jennings, N.R. 1996, "Negotiation through argumentation – a preliminary report". In *Proceedings of the 2nd International Conference on Multi-Agent Systems (ICMAS-96),* Kyoto, Japan, pp. 267-274.

[70] Parsons, S., Sierra, C. and Jennings, N.R. 1998, "Multi-context argumentative agents". In *4th Symposium on Logical Formalizations of Common Sense Reasoning*, pp. 298-349.

[71] Parsons, S., Sierra, C.A. and Jennings, N.R. 1998, "Agents that reason and negotiate by arguing". *Journal of Logic and Computation*, **8**(3), pp. 261-292.

[72] Rahwan, I., McBurney, P. and Sonenberg L. 2003, "Towards a Theory of Negotiation Strategy (A Preliminary Report)". In *Proceedings of the AAMAS Workshop on Game Theoretic and Decision Theoretic Agents (GTDT)*, 14 July, Australia.

[73] Rahwan. I., Sonenberg, L., Jennings, N. R. and McBurney, P. 2003, "STRATUM: A Methodology for Designing Heuristic Agent Negotiation Strategies". In *Applied Artificial Intelligence*, vol. 21, no. 6, pp. 489-527.

[74] Rahwan, I. and Tohmé, I. (2010), "Collective Argument Evaluation as Judgement Aggregation", In *Proc. of 9th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2010)*, van der Hoek, Kaminka, Lespérance, Luck and Sen (eds.), May, 10–14, 2010, Toronto, Canada, pp. 417-424.

[75] Reed, C. and Long, D. 1997, "Ordering and focusing in an architecture for persuasive discourse planning". In *Proceedings of the 6th European Workshop on Natural Language Generation,* Duisburg, Germany.

[76] Reed, C. 1998, "Dialogue frames in agent communication". In *Proceedings of the 3rd International Conference on Multi-Agent Systems (ICMAS-98),* Paris, France, pp. 246-253.

[77] Rosenschein, S.J. and Zlotkin, G. 1994, *Rules of Encounter: Designing Conventions for Automated Negotiation among Computers*. MIT Press, Cambridge, CA.

[78] Roy, B. 1968, Classement et choix en présence de points de vue multiple (La méthode ELECTRE), *R.I.R.O.* **8**, pp. 57-75.

[79] Roy, B. 1985, *Méthodologie Multicritère d'Aide à la Décision*. Economica, Paris.

[80] Roy, B. 1996, *Multicriteria Methodology for Decision Aiding*. Dordrecht, Kluwer Academic.
[81] Saaty, T. 1980, *The Analytic Hierarchy Process*. New York, McGraw-Hill.
[82] Salo, A. A. 1995, "Interactive Decision Aiding for Group Decision Support", *European Journal of Operational Research* 84, pp. 134-149.
[83] Schroeder, Michael 1999, "An efficient argumentation framework for negotiating autonomous agents". In *Proceedings of Modelling Autonomous Agents in a Multi-Agent World MAAMAW99*, LNAI1647, Springer-Verlag .
[84] Shakun, M. F. 1988, *Evolutionary Systems Design*. Oakland, CA, Holden-Day.
[85] Shakun, M. F. 1991, "Airline Buyout: Evolutionary Systems Design and Problem Restructuring in Group Decision and Negotiation", *Management Science* **37**(10), pp. 1291-1303.
[86] Sierra, C., Jennings, N.R., Noriega, P., Parsons, S. 1998, A framework for argumentation-based negotiation. In *Proc of ATAL,* pp. 167-182 Springer-Verlag 1997.
[87] Sillince, John A.A. 1993, "Multi-agent conflict resolution: a computational framework for an intelligent argumentation program". *Knowledge-Based Systems,* Vol. 7, no. 2 June 1994. Butterworth-Heinemann Ltd.
[88] Siskos, J., Spyridakos, A. and Yannacopoulos, D. 1993, "MINORA: A Multicriteria Decision Aiding System for Discrete Alternatives," *Journal of Information Science and Technology* **2**(2), pp. 136-149.
[89] Siskos, Y. and Yannakopoulos, D. 1985, "UTASTAR, An Ordinal Regression Method for Building Additive Value Functions," *Investigacao Operacional* **5**(1), pp. 39-53.
[90] Siskos, Y., Grigoroudis, E., Zopounidis, C. and Saurais, O. 1997, "Measuring Customer Satisfaction Using a Collective Preference Disaggregation Model", *Journal of Global Optimization* **12**, pp. 175-195.
[91] Sprague, R.H. and Carlson, E. D. 1982, *Building effective decision support systems.* Englewood Cliffs, N.J., Prentice-Hall.
[92] Stanoulov, N. 1995, "A Parsimonious Outranking Method for Individual and Group Decisionmaking and its Computerized Support," *IEEE Transactions on Systems*, *Man*, *and Cybernetics* **25**(2), pp. 266-276.
[93] Stanoulov, N. 1994, "Expert Knowledge and Computer-aided Group Decision Making: Some Pragmatic Reflections," *Annals of Operations Research* 51, pp. 141-162.
[94] Sycara, K.P. 1989a, "Argumentation: planning other agents' plans". In *Proceedings of the 11th International Joint Conference on Artificial Intelligence (IJCAI-89), Detroit, MI,* pp. 517-523.
[95] Sycara, K.P. 1989b, "Multiagent compromise via negotiation". In *Distributed Artificial Intelligence* (eds. L. Gasser and M. Huhns), Vol. II, pp. 119-138. Pitman, London and Morgan Kaufmann, San Mateo, CA.
[96] Turban, E. 1988, *Decision Support Systems and Expert Systems*: *Managerial Perspectives*. NewYork, Macmillan.
[97] Turban, E. 1995, *Decision Support and Expert Systems: Management Support Systems*. Englewood Cliffs, N.J., Prentice-Hall.
[98] Turnovec, F. 2002, "Decision Making Games in the European Union". In *Proceedings of the 37th Annual Conference of the Operational Research Society*, New Zealand, University of Auckland, pp. 245-254.

[99] Uno, T. 2003, Efficient Computation of Power Indices for Weighted Majority Games, NII Technical Report, National Institute of Informatics.

[100] Van Houtven, L. 2002, *Governance of the IMF. Decision Making, Institutional Oversight, Transparency, and Accountability*, Pamphlet Series, No. 53, International Monetary Fund.

[101] Verheij, B. 1996, *Two approaches to dialectical argumentation: admissible sets and argumentation stages*. Department of Metajuridica, University of Limburg, Maastricht.

[102] Vetchera, R. 1991, "Integrating Databases and Preference Evaluations in Group Decision Support – A Feedback-Oriented Approach," *Decision Support Systems* 7, pp. 67-77.

[103] Walton, D.N. and Krabbe, E.C.W. 1995, *Commitment in Dialogue: Basic Concepts of Interpersonal Reasoning.* State University of New York Press, Albany, NY.

[104] Wooldridge, M. and Jennings, N.R. 1995, Intelligent agents: theory and practice. *The Knowledge Engineering Review*, **10**(2), 115-152.

[105] Wooldridge, Michael 2002, *An introduction to multiagent systems*. John Wiley & Sons Ltd.

[106] Wooldridge, M., McBurney, P. and Parsons, S. 2006, On the Meta-logic of Arguments, in *Postproceedings of ArgMAS 2005*, LNAI 4049, Springer-Verlag Berlin Heidelberg, pp. 42-56.

[107] European Personnel Selection Office (EPSO) – Selection Procedures: http://europa.eu/epso/discover/selection_proced/selection/index_en.htm (retrieved 10/4/2011)

Appendix

Source Code

UTASTAR Implementation (class com.gorbas.UTASTAR)

```
package com.gorbas.utastar;

import com.gorbas.common.Logging;
import com.gorbas.control.AbstractController;
import com.gorbas.model.AbstractEntity;
import com.gorbas.model.Alternative;
import com.gorbas.model.Characteristic;
import com.gorbas.model.Contest;
import com.gorbas.model.Criterion;
import com.gorbas.model.DecisionMaker;
import com.gorbas.model.DecisionMakerAlternativeRank;
import com.gorbas.model.DecisionMakerAlternativeUsage;
import com.gorbas.model.DmAlternativeCriterionUtilityValue;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import javax.persistence.NoResultException;
import org.apache.commons.math.linear.RealVector;
import org.apache.commons.math.optimization.GoalType;
import org.apache.commons.math.optimization.RealPointValuePair;
import org.apache.commons.math.optimization.linear.LinearConstraint;
import org.apache.commons.math.optimization.linear.LinearObjectiveFunction;
import org.apache.commons.math.optimization.linear.Relationship;
import org.apache.commons.math.optimization.linear.SimplexSolver;

/**
 *
 *
 */
public class UTASTAR {

    static int COLUMN_LENGTH = 10;
    static Logging LOGGER = Logging.get(UTASTAR.class);

    static void sysout(String str) {
        System.out.println(str);
    }
```

```
    static void sysout1(String str) {
        System.out.print(str);
    }
    private final Map<DecisionMaker, Map<Alternative, Integer>>
rankPerAlternativePerDecisionMaker = new HashMap<DecisionMaker, Map<Alternative,
Integer>>();

    public UTASTAR(Contest contest) {
        //Retrieve ranks and initiate rank map
        boolean newEntityManger = !AbstractController.inSession();
        if (newEntityManger) {
            AbstractController.startEntityManager();
        }
        try {
            boolean newTransaction = !AbstractController.inTransaction();
            if (newTransaction) {
                AbstractController.beginTransaction();
            }
            try {
                LOGGER.trace("Fill valuePerCriterionPerAlternative - BEGIN");
                /**
                 *
                 */
                Map<Alternative, Map<Criterion, Double>>
valuePerCriterionPerAlternative = new HashMap<Alternative, Map<Criterion,
Double>>();
                List<Characteristic> characteristics =
Characteristic.retrieveCharacteristics(contest);
                for (Characteristic characteristic : characteristics) {
                    if (!
valuePerCriterionPerAlternative.containsKey(characteristic.getAlternative())) {

valuePerCriterionPerAlternative.put(characteristic.getAlternative(), new
HashMap<Criterion, Double>());
                    }
                    Map<Criterion, Double> weightPerCriterion =
get(valuePerCriterionPerAlternative, characteristic.getAlternative());
                    LOGGER.info("weightPerCriterion=weightPerCriterion for
alternative=" + characteristic.getAlternative());
                    weightPerCriterion.put(characteristic.getCriterion(),
characteristic.getValue());
                    LOGGER.info("weightPerCriterion put (criterion=" +
characteristic.getCriterion() + " value=" + characteristic.getValue() + ")");
                }
                LOGGER.trace("Fill valuePerCriterionPerAlternative - END");

                LOGGER.trace("Keep ranking of alternatives for each decision
maker - BEGIN");
                /**
                 * Keep ranking of alternatives for each decision maker
                 */
                List<DecisionMakerAlternativeRank> decisionMakerAlternativeRanks
= DecisionMakerAlternativeRank.retrieveDecisionMakerAlternativeRank(contest);
                for (DecisionMakerAlternativeRank decisionMakerAlternativeRank :
decisionMakerAlternativeRanks) {
                    if (get(
                            rankPerAlternativePerDecisionMaker,
                            decisionMakerAlternativeRank.getDecisionMaker()) ==
null) {//Εαν δεν υπάρχει ήδη Map για τα στοιχεια του τρέχοντος αποφασίζοντα
                        rankPerAlternativePerDecisionMaker.put(
                                decisionMakerAlternativeRank.getDecisionMaker(),
                                new HashMap<Alternative, Integer>());//το
προσθέτουμε στο κεντρικο map
                    }
```

```
                    get(rankPerAlternativePerDecisionMaker,
decisionMakerAlternativeRank.getDecisionMaker()).
                            put(
                            decisionMakerAlternativeRank.getAlternative(),

decisionMakerAlternativeRank.getUsageValue());//προσθετουμε στο map του τρέχοντος
αποφασιζοντα, την επιδοση που έδωσε για την τρέχουσα εναλλακτική
                }
                LOGGER.trace("Keep ranking of alternatives for each decision
maker - END");

                LOGGER.trace("Retrieve criteria");
                List<Criterion> criteria = Criterion.retrieveCriteria(contest);
                LOGGER.trace("Retrieved criteria are " + criteria.size());

                LOGGER.trace("prodiataxi - BEGIN");
                prodiataxi(valuePerCriterionPerAlternative,
rankPerAlternativePerDecisionMaker, criteria);
                LOGGER.trace("prodiataxi - END");
                if (newTransaction) {
                    AbstractController.commit();
                }
            } catch (Exception e) {
                if (newTransaction) {
                    AbstractController.rollback();
                }
                throw new RuntimeException(e);
            }
        } finally {
            if (newEntityManger) {
                AbstractController.closeEntityManager();
            }
        }

    }

    private void prodiataxi(
            Map<Alternative, Map<Criterion, Double>>
valuePerCriterionPerAlternative,
            Map<DecisionMaker, Map<Alternative, Integer>>
rankPerAlternativePerDecisionMaker,
            List<Criterion> criteria) {
        for (Map.Entry<DecisionMaker, Map<Alternative, Integer>> entry :
rankPerAlternativePerDecisionMaker.entrySet()) {
            Map<Alternative, Integer> rankPerAlternative = entry.getValue();
            List<AlternativeWithRank> alternatives = new
ArrayList<AlternativeWithRank>();
            LOGGER.info("PRODIATAXI");
            for (Map.Entry<Alternative, Integer> _entry :
rankPerAlternative.entrySet()) {
                alternatives.add(new AlternativeWithRank(_entry.getKey(),
_entry.getValue()));
                LOGGER.info(_entry.getKey() + "->" + _entry.getValue());
            }
            AlternativeWithRank[] tmp = alternatives.toArray(new
AlternativeWithRank[0]);
            Arrays.sort(tmp);
            utastark200(entry.getKey(), tmp, valuePerCriterionPerAlternative,
criteria);
        }
    }

    private static void saveUtilityValuePerCriterionPerAlternative(DecisionMaker
dm, List<Criterion> criteria, Map<Alternative, Map<Criterion, double[]>> w,
double[] averageCriterionWeight) {
```

```
        LOGGER.trace("saveUtilityValuePerCriterionPerAlternative(dm=" + dm +
",criteria=" + criteria + ", w=" + w + ",averageCriterionWeight=" +
averageCriterionWeight + ") BEGIN");
        for (Map.Entry<Alternative, Map<Criterion, double[]>> entry :
w.entrySet()) {
            int index = 0;
                    double altUtility = 0.0;
            for (int a = 1; a <= criteria.size(); a++) {
                        Criterion cr = criteria.get(a - 1);
                        double[] wOfCr = entry.getValue().get(cr);
                double utilityValue = 0.0;
                        for (int d = 0; d < cr.getNumberOfDiscreteValues() -
1; d++, index++)
                                utilityValue += (averageCriterionWeight[index] *
wOfCr[d]);

                        altUtility += utilityValue;

                        DmAlternativeCriterionUtilityValue dmcuv = new
DmAlternativeCriterionUtilityValue();
                dmcuv.setAlternative(entry.getKey());
                dmcuv.setCriterion(cr);
                dmcuv.setDecisionMaker(dm);
                dmcuv.setUtilityValue(utilityValue);

                try {
                    DmAlternativeCriterionUtilityValue _dmcuv =
DmAlternativeCriterionUtilityValue.retrieveByAlternativeDecisionMakerCriterion(dm
cuv.getAlternative(), dmcuv.getDecisionMaker(), dmcuv.getCriterion());
                    if (_dmcuv != null) {
                        _dmcuv.setUtilityValue(dmcuv.getUtilityValue());
                        dmcuv = _dmcuv;
                    }
                } catch (Exception e) {
                    LOGGER.error("saveUtilityValuePerCriterionPerAlternative()\t"
+ e.getMessage(), e);
                }
                if (dmcuv.getId() == null) {
                    AbstractController.persist(dmcuv);
                } else {
                    AbstractController.merge(dmcuv);
                }
            }

                    // Save alternative utility (sum of all criteria utilities of
the alternative).
                    DecisionMakerAlternativeUsage dmau = null;
                    try { dmau = DecisionMakerAlternativeUsage.retrieve(dm,
entry.getKey()); }
                    catch (NoResultException ex) { }
                    if (null == dmau) {
                            dmau = new DecisionMakerAlternativeUsage();
                            dmau.setDecisionMaker(dm);
                            dmau.setAlternative(entry.getKey());
                    }
                    dmau.setUsageValue(altUtility);
                    if (null == dmau.getId()) AbstractController.persist(dmau);
                    else AbstractController.merge(dmau);
        }
        LOGGER.trace("saveUtilityValuePerCriterionPerAlternative() END");
    }
    private static final double DELTA = 0.05;

       private static void printConstraint(LinearConstraint c) {
              RealVector coeffs = c.getCoefficients();
```

```
            int dim = coeffs.getDimension();
            for (int i = 0; i < dim; i++)
                  System.out.printf("%8lf |", coeffs.getEntry(i));
            String rel = "?";
            switch(c.getRelationship()) {
                  case EQ: rel = "= "; break;
                  case GEQ: rel = ">="; break;
                  case LEQ: rel = "<="; break;
            }
            System.out.print(rel + " |");
            System.out.printf("%8lf |", c.getValue());
      }

      private static void printObjFunction(LinearObjectiveFunction f) {
            RealVector coeffs = f.getCoefficients();
            int dim = coeffs.getDimension();
            for (int i = 0; i < dim; i++)
                  System.out.printf("%8lf |", coeffs.getEntry(i));
            System.out.printf("%8lf |", f.getConstantTerm());
      }

    /**
     *
     * @param sortedAlternatives
     * @param valuePerCriterionPerAlternative
     * @param criteria
     */
    private static void utastark200(
            DecisionMaker dm,
            AlternativeWithRank[] sortedAlternatives,
            Map<Alternative, Map<Criterion, Double>>
valuePerCriterionPerAlternative,
            List<Criterion> criteria) {
        {
            StringBuilder sortedAlternativesAsStr = new StringBuilder();
            for (AlternativeWithRank awr : sortedAlternatives) {
                sortedAlternativesAsStr.append(awr.alternative).append("-
>").append(awr.rank).append(", ");
            }
            StringBuilder valuePerCriterionPerAlternativeStr = new
StringBuilder();
            for (Map.Entry<Alternative, Map<Criterion, Double>> e :
valuePerCriterionPerAlternative.entrySet()) {

valuePerCriterionPerAlternativeStr.append("Alternative[").append(e.getKey()).appe
nd("]{");
                for (Map.Entry<Criterion, Double> e1 : e.getValue().entrySet()) {

valuePerCriterionPerAlternativeStr.append(e1.getKey()).append("=").append(e1.getV
alue()).append(",");
                }
                valuePerCriterionPerAlternativeStr.append("}, ");
            }
            StringBuilder criteriaStr = new StringBuilder();
            for (Criterion c : criteria) {
                criteriaStr.append(c).append(",");
            }

            LOGGER.trace(
                    "utastark200 for dm=" + dm + " sortedAlternatives=" +
sortedAlternativesAsStr + " valuePerCriterionPerAlternative=" +
valuePerCriterionPerAlternativeStr + " criteria=" + criteriaStr);
        }
        int numberOfAlternatives = sortedAlternatives.length;
        int numberOfW = 0;
```

```
        for (Criterion criterion : criteria) {
            numberOfW += criterion.getNumberOfDiscreteValues();
        }

        LOGGER.trace("retrieveStepsPerCriterion()");
        Map<Criterion, double[]> stepsPerCriterion =
retrieveStepsPerCriterion(valuePerCriterionPerAlternative, criteria);
        LOGGER.trace("calcualteU()");
        Map<Alternative, Map<Criterion, double[]>> u =
calcualteU(valuePerCriterionPerAlternative, stepsPerCriterion);
        LOGGER.trace("calculateW()");
        Map<Alternative, Map<Criterion, double[]>> w = calculateW(u,
stepsPerCriterion);
        LOGGER.trace("calculateA1P()");
        Map<Alternative, double[]> A1P = calculateA1P(w, sortedAlternatives,
criteria);
        LOGGER.trace("calculateA2P()");
        Map<Alternative, double[]> A2P = calculateA2P(sortedAlternatives,
criteria);
        LOGGER.trace("combineA1PandA2P() [keep it as At]");
        Map<Alternative, double[]> At = combineA1PandA2P(A1P, A2P);

        for (Map.Entry<Alternative, double[]> e : At.entrySet()) {
            sysout1("\n" + e.getKey().getDescription() + "\t|");
            for (Double a : e.getValue()) {
                sysout1("" + a + "\t|");
            }
        }
        Map<Alternative, double[]> AEQ = retrieveEqualWithNext(At,
sortedAlternatives);//Οι εναλλακτικές που έχουν ίδια προτεραιότητα με την επόμενη
εναλλακτική
        Map<Alternative, double[]> AA = retrieveNotEqualWithNext(At,
sortedAlternatives);//Οι εναλλακτικές που δεν έχουν ιδια προτεραιότητα με την
επόμενη εναλλακτική


        /**
         * Πίνακας που θα έχει για όλες τις εναλλακτικές με την σειρά που τις
έχει στο sortedAlternatives
         */
        double[][] AtAsArray = new double[sortedAlternatives.length]
[At.values().iterator().next().length];
        double[] b = new double[sortedAlternatives.length];
        for (int i = 0; i < sortedAlternatives.length; i++) {
            Alternative currentAlternative = sortedAlternatives[i].alternative;
            double[] valueArray = null;
            if (AEQ.containsKey(currentAlternative)) {
                valueArray = get(AEQ, currentAlternative);
                b[i] = 0.0;
            } else {
                valueArray = get(AA, currentAlternative);
                b[i] = DELTA;
            }
                    System.arraycopy(valueArray, 0, AtAsArray[i], 0,
valueArray.length);
        }
             b[b.length - 1] = 1.0; // Σw_i = 1.0

        int numberOfVariables = AtAsArray[0].length;
        /**
         *
         * Το πλήθος των μεταβλητών w του προβλήματος είναι όσο είναι το άθροισμα
του πλήθους των διακριτών τιμών των κριτηρίων
         * Μείον το πλήθος των κριτηρίων, καθώς η μικρότερη τιμή της κλιμακας
ειναι γνωστό οτι έχει τιμή 0
```

```
         *
         * Οι μεταβλητές σίγμα είναι σε πλήθος τόσες όσες 2 φορές το πλήθος των
εναλλακτικών
         *
         * Για τις μεν μεταβλητές w αποδίδω τιμή μηδέν '0' στο διάνυσμα F, ενώ
για τα σίγμα τοποθετώ τιμή ένα '1'
         */
        int summaryOfDiscreteValues = 0;
        for (Criterion criterion : criteria) {
            summaryOfDiscreteValues += criterion.getNumberOfDiscreteValues() - 1;
        }
        double[] F = new double[numberOfVariables -
criteria.size()];//summaryOfDiscreteValues + 2 * sortedAlternatives.length];
        for (int i = summaryOfDiscreteValues; i < F.length; i++) {
            F[i] = 1.0;
        }

        sysout("Number of variables is " + numberOfVariables + " while the F has
length " + F.length);

        try {

                    /*
                     * UTASTAR Step 3.
                     */

                    // describe the optimization problem
                    LinearObjectiveFunction f = new LinearObjectiveFunction(F,
0.0);
                    ArrayList<LinearConstraint> constraints = new
ArrayList<LinearConstraint>();

                    sysout("Constraints:");
                    sysout("                                    ".substring(0,
COLUMN_LENGTH));
                    for (int i = 1; i <= criteria.size(); i++) {
                           for (int j = 1; j <= criteria.get(i -
1).getNumberOfDiscreteValues() - 1; j++) {
                                  sysout1(("|" + criteria.get(i -
1).getDescription() + "            ").substring(0, COLUMN_LENGTH));
                           }
                    }
                    sysout1("\n                              ".substring(0,
COLUMN_LENGTH));
                    for (int i = 1; i <= criteria.size(); i++) {
                           for (int j = 1; j <= criteria.get(i -
1).getNumberOfDiscreteValues() - 1; j++) {
                                  sysout1(("|w" + i + j + "
").substring(0, COLUMN_LENGTH));
                           }
                    }
                    for (AlternativeWithRank a : sortedAlternatives) {
                           sysout1(("|c" + a.alternative.getDescription() + "-
").substring(0, COLUMN_LENGTH));
                           sysout1(("|c" + a.alternative.getDescription() + "+
").substring(0, COLUMN_LENGTH));
                    }


                    for (int indexInArray = 0; indexInArray <
sortedAlternatives.length; indexInArray++) {//Each alternative adds a constraint
                           AlternativeWithRank _entry =
sortedAlternatives[indexInArray];
                           Alternative entry = _entry.alternative;
```

```
                        sysout1(("\n" + _entry.rank + "->" +
entry.getDescription() + "(" + entry.getId() + ")" + "          ").substring(0,
COLUMN_LENGTH));

                        double[] atValues = get(At, entry);
                        double[] criteriaValues = new double[atValues.length -
criteria.size()];
                        int indexInAt = 0;
                        int indexInCriteriaValues = 0;
                        for (int i = 1; i <= criteria.size(); i++) {
                              for (int j = 1; j <= criteria.get(i -
1).getNumberOfDiscreteValues() - 1; j++) {
                                    criteriaValues[indexInCriteriaValues++] =
atValues[indexInAt];
                                    sysout1(("|" +
criteriaValues[indexInCriteriaValues - 1] + "           ").substring(0,
COLUMN_LENGTH));
                                    indexInAt++;
                              }
                              indexInAt++;
                        }
                        for (; indexInAt < atValues.length; indexInAt++) {
                              criteriaValues[indexInCriteriaValues++] =
atValues[indexInAt];
                              sysout1(("" +
criteriaValues[indexInCriteriaValues - 1] + "           ").substring(0,
COLUMN_LENGTH));
                        }

                        double value = b[indexInArray];
                        sysout1("" + (value == DELTA ? ">=" : "=") + value);

                        constraints.add(new LinearConstraint(criteriaValues,
value == DELTA ? Relationship.GEQ : Relationship.EQ, value));
                  }

                  sysout1(("\n               ").substring(0, COLUMN_LENGTH));
                  for (double _f : F) {
                        sysout1("|-------------------------".substring(0,
COLUMN_LENGTH));
                  }
                  sysout1("\n                           ".substring(0,
COLUMN_LENGTH));
                  for (double _f : F) {
                        sysout1(("|" + _f + "
").substring(0, COLUMN_LENGTH));
                  }
                  sysout("\n\n\n");

                  // create and run the solver
                  SimplexSolver solver = new SimplexSolver();
//            solver.setMaxIterations(10);
                  RealPointValuePair solution = solver.optimize(f, constraints,
GoalType.MINIMIZE, true);

                  Map<Criterion, double[]> stepValuePerCriterion =
retrieveStepsPerCriterion(valuePerCriterionPerAlternative, criteria);
                  // get the solution
                  sysout("Result's length=" + solution.getPointRef().length);
                  int index = 0;
                  for (int i = 1; i <= criteria.size(); i++) {
                        for (int j = 1; j <= criteria.get(i -
1).getNumberOfDiscreteValues(); j++) {
```

```
                                    sysout("\t" + criteria.get(i - 1).getName() +
"#" + j + "[" + stepValuePerCriterion.get(criteria.get(i - 1))[j - 1] + "]" + "-
>" + solution.getPoint()[index]);
                                    index++;
                             }
                      }
                      for (int i = 0; index < solution.getPointRef().length; i++) {
                             sysout("\tc" + i + "->" + solution.getPointRef()
[index]);
                             index++;
                      }


                      sysout("Min=" + solution.getValue());


                      /*
                       * UTASTAR Step 4 (post-optimality analysis): Test for
multiple near-optimal solutions in step 3 and use the average of all of them.
                       */

                      // If Σσ = 0 and every sigma value is also zero, then totally
remove the zeroes from the linear program and do not add a sigma constraint.
                      boolean noSigma = solution.getValue() == 0;
                      if (noSigma) {
                             double[] sol = solution.getPoint();
                             for (int i = numberOfW - criteria.size(); i <
sol.length; i++) {
                                    if (sol[i] != 0.0) {
                                           noSigma = false;
                                           break;
                                    }
                             }
                      }

                      if (noSigma) {
                             // Remove the sigma part from constraints.
                             for (int i = 0; i < constraints.size(); i++) {
                                    LinearConstraint lc = constraints.get(i);
                                    lc = new
LinearConstraint(lc.getCoefficients().getSubVector(0, numberOfW -
criteria.size()), lc.getRelationship(), lc.getValue());
                                    constraints.set(i, lc);
                             }
                      }
                      else {
                             // Add the constraint Σσ <= z* + ε to the system.
                             double[] sigmaConstraint = new double[F.length];
                             System.arraycopy(solution.getPointRef(), numberOfW -
criteria.size(),
                                           sigmaConstraint, numberOfW -
criteria.size(), sortedAlternatives.length * 2);
                             double sv = solution.getValue();
                             constraints.add(new LinearConstraint(sigmaConstraint,
Relationship.LEQ, sv < 5e-11 ? sv + 5e-11 : sv * 1.1 /*sv + ε*/));
                      }

                      // Now obtain the solutions by trying to maximize the
criteria utility functions (sum of value weights of each criterion).
                      RealPointValuePair[] nearOptimalSolutions = new
RealPointValuePair[criteria.size()];
                      int wIndex = 0;
                      int cIndex = 0;
                      for (Criterion c : criteria) {
                             Arrays.fill(F, 0.0);
```

```
                        int nrWeights = c.getNumberOfDiscreteValues() - 1;
                        for (int i = 0; i < nrWeights; i++, wIndex++)
                              F[wIndex] = 1.0;
                        nearOptimalSolutions[cIndex++] = solver.optimize(new
LinearObjectiveFunction(F, 0.0), constraints, GoalType.MAXIMIZE, true);
                  }

                  // Now calculate the mean value of the solutions (the
weights).
                  double[] avgSolution = new double[numberOfW -
criteria.size()];
                  for(int i = 0; i < nearOptimalSolutions.length; i++) {
                        double[] p = nearOptimalSolutions[i].getPointRef();
                        for (int j = 0; j < avgSolution.length; j++)
                              avgSolution[j] += p[j];
                  }
                  for (int i = 0; i < avgSolution.length; i++)
                        avgSolution[i] /= nearOptimalSolutions.length;

                  double[] discreteValueWeights = new double[numberOfW];
                  int dvi = 0;
                  int si = 0;
                  for (Criterion c : criteria) {
                        discreteValueWeights[dvi++] = 0;
                        for (int i = 1; i < c.getNumberOfDiscreteValues(); i+
+, dvi++, si++)
                              discreteValueWeights[dvi] =
discreteValueWeights[dvi - 1] + avgSolution[si];
                  }


            sysout("");
            saveUtilityValuePerCriterionPerAlternative(dm, criteria, w,
avgSolution);

            //εκτύπωση AverageUtilityValueForDecisionMakerCriterion
            sysout("\nAverageUtilityValueForDecisionMakerCriterion");
            for (Criterion c : criteria) {
                double utilityValue =
DmAlternativeCriterionUtilityValue.AverageUtilityValueForDecisionMakerCriterion(d
m, c);
                sysout(c.getName() + "->" + utilityValue);
            }

            sysout("\nDecisionMakerAlternativeUsage");
            for (Alternative a : At.keySet()) {
                DecisionMakerAlternativeUsage usage =
DecisionMakerAlternativeUsage.retrieve(dm, a);
                sysout(a.getDescription() + "->" + usage.getUsageValue());
            }

        } catch (Exception e) {
            e.printStackTrace();
            throw new RuntimeException(e);
        }
    }

    /**
     *
     * @param values
     * @return μέσο ορο των τιμών που βρισκονται στην συλλογη "values"
     */
    private static Double avg(java.util.Collection<Double> values) {
        if (values.isEmpty()) {
            return 0.0;
```

```
        }
        double avg = 0.0;
        for (double v : values) {
            avg += v;
        }
        return avg / values.size();
    }

    /**
     *
     * @param At
     * @param sortedAlternatives
     * @return Πίνακα για τις εναλλακτικές που δεν έχουν ιδια προτεραιότητα με
την επόμενη τους. Να σημειωθεί οτι η τελευταία δεν ανήκει σε αυτόν τον πίνακα
     */
    private static Map<Alternative, double[]>
retrieveNotEqualWithNext(Map<Alternative, double[]> At, AlternativeWithRank[]
sortedAlternatives) {
        Map<Alternative, double[]> AAp = new HashMap<Alternative, double[]>();
        for (int i = 0; i < sortedAlternatives.length - 1; i++) {
            AlternativeWithRank currentAlternativeWithRank =
sortedAlternatives[i];
            AlternativeWithRank nextAlternativeWithRank = sortedAlternatives[i +
1];
            if (currentAlternativeWithRank.rank != nextAlternativeWithRank.rank)
{
                AAp.put(currentAlternativeWithRank.alternative, get(At,
currentAlternativeWithRank.alternative));
            }

        }
        return AAp;
    }

    /**
     *
     * @param At
     * @param sortedAlternatives
     * @return Πίνακα για τις εναλλακτικές που έχουν ιδια προτεραιότητα με την
επόμενη τους. Να σημειωθεί ότι και η εναλλακτική που είναι τελευταία ανήκει σε
αυτή την ομάδα
     */
    private static Map<Alternative, double[]>
retrieveEqualWithNext(Map<Alternative, double[]> At, AlternativeWithRank[]
sortedAlternatives) {
        Map<Alternative, double[]> AEQ = new HashMap<Alternative, double[]>();
        for (int i = 0; i < sortedAlternatives.length - 1; i++) {
            AlternativeWithRank currentAlternativeWithRank =
sortedAlternatives[i];
            AlternativeWithRank nextAlternativeWithRank = sortedAlternatives[i +
1];
            if (currentAlternativeWithRank.rank == nextAlternativeWithRank.rank)
{
                AEQ.put(currentAlternativeWithRank.alternative, get(At,
currentAlternativeWithRank.alternative));
            }

        }
        Alternative lastAlternative =
sortedAlternatives[sortedAlternatives.length - 1].alternative;
        AEQ.put(lastAlternative, get(At, lastAlternative));
        return AEQ;
    }
```

```
    private static Map<Alternative, double[]> combineA1PandA2P(Map<Alternative,
double[]> A1P, Map<Alternative, double[]> A2P) {
        Map<Alternative, double[]> At = new HashMap<Alternative, double[]>();
        for (Alternative alternative : A1P.keySet()) {
            double[] A1Pv = get(A1P, alternative);
            double[] A2Pv = get(A2P, alternative);
            double[] values = new double[A1Pv.length + A2Pv.length];
                    System.arraycopy(A1Pv, 0, values, 0, A1Pv.length);
                    System.arraycopy(A2Pv, 0, values, A1Pv.length, A2Pv.length);

            At.put(alternative, values);
        }

        return At;
    }

    /**
     * Συντελεστές σ
     * @param w
     * @param sortedAlternativeWithRanks
     * @param criteria
     * @return
     */
    private static Map<Alternative, double[]> calculateA2P(AlternativeWithRank[]
sortedAlternativeWithRanks, List<Criterion> criteria) {
        Map<Alternative, double[]> A2P = new HashMap<Alternative, double[]>();
        for (int a = 0; a < sortedAlternativeWithRanks.length; a++) {
            Alternative alternative = sortedAlternativeWithRanks[a].alternative;
                    double[] sigmas = new
double[sortedAlternativeWithRanks.length * 2]; // Initialized to zero.
            A2P.put(alternative, sigmas);

            if (a < (sortedAlternativeWithRanks.length - 1)) {
                sigmas[a * 2] = 1.0;
                sigmas[a * 2 + 1] = -1.0;
                sigmas[a * 2 + 2] = -1.0;
                sigmas[a * 2 + 3] = 1.0;
            }

        }
        return A2P;
    }

    /**
     * Συντελεστής για την βαρύτητα κάθε διακριτής τιμής κάθε κριτηρίου για κάθε
εναλλακτική
     * @param w
     * @param sortedAlternativeWithRanks
     * @param criteria
     * @return
     */
    private static Map<Alternative, double[]> calculateA1P(Map<Alternative,
Map<Criterion, double[]>> w, AlternativeWithRank[] sortedAlternativeWithRanks,
List<Criterion> criteria) {
        Map<Alternative, double[]> A1P = new HashMap<Alternative, double[]>();
        int numberOfTotalDiscreteValues = 0;
        for (Criterion criterion : criteria) {
            numberOfTotalDiscreteValues += criterion.getNumberOfDiscreteValues();
        }
        for (int a = 0; a < sortedAlternativeWithRanks.length - 1; a++) {
            Alternative alternative = sortedAlternativeWithRanks[a].alternative;
            Alternative nextAlternative = sortedAlternativeWithRanks[a +
1].alternative;
```

```
                    double[] allWeights = new
double[numberOfTotalDiscreteValues]; // Initialized to 0.
            A1P.put(alternative, allWeights);

            int currentStep = 0;
            for (int c = 0; c < criteria.size(); c++) {
                Criterion criterion = criteria.get(c);
                          double[] weights = get(get(w, alternative), criterion);
                          double[] nextAltWeights = get(get(w, nextAlternative),
criterion);
                for (int step = 0; step < criterion.getNumberOfDiscreteValues();
step++, currentStep++) //TODO:Check§
                    allWeights[currentStep] = weights[step] -
nextAltWeights[step];
            }
        }
        Alternative lastAlternative =
sortedAlternativeWithRanks[sortedAlternativeWithRanks.length - 1].alternative;
        A1P.put(lastAlternative, new double[numberOfTotalDiscreteValues]);
        for (int i = 0; i < get(A1P, lastAlternative).length; i++) {
            get(A1P, lastAlternative)[i] = 1.0;
        }

        return A1P;
    }

    private static Map<Alternative, Map<Criterion, double[]>>
calculateW(Map<Alternative, Map<Criterion, double[]>> u, Map<Criterion, double[]>
stepsPerCriterion) {
        Map<Alternative, Map<Criterion, double[]>> w = new HashMap<Alternative,
Map<Criterion, double[]>>();
        for (Map.Entry<Alternative, Map<Criterion, double[]>> entry0 :
u.entrySet()) {
            Alternative alternative = entry0.getKey();
            w.put(alternative, new HashMap<Criterion, double[]>());
            for (Map.Entry<Criterion, double[]> entry1 :
entry0.getValue().entrySet()) {
                Criterion criterion = entry1.getKey();
                double[] value = entry1.getValue();
                double[] zeros = new
double[criterion.getNumberOfDiscreteValues()];

                get(w, alternative).put(criterion, zeros);
                double[] steps = get(stepsPerCriterion, criterion);
                for (int stepIndex = 0; stepIndex < (steps.length - 1);
stepIndex++) {
                    if (value[stepIndex] == 0) {
                        get(get(w, alternative), criterion)[stepIndex] = 1.0;
                    } else if (value[stepIndex] == 1) {
                        break;
                    } else {
                        get(get(w, alternative), criterion)[stepIndex] =
get(get(u, alternative), criterion)[stepIndex + 1];
                        break;

                    }
                }
            }
        }
        return w;
    }

    /**
     *
     *
```

```
     * @param valuePerCriterionPerAlternative
     * @param stepsPerCriterion
     * @return
     */
    private static Map<Alternative, Map<Criterion, double[]>>
calcualteU(Map<Alternative, Map<Criterion, Double>>
valuePerCriterionPerAlternative, Map<Criterion, double[]> stepsPerCriterion) {
        Map<Alternative, Map<Criterion, double[]>> u = new HashMap<Alternative,
Map<Criterion, double[]>>();
        for (Map.Entry<Alternative, Map<Criterion, Double>> entry0 :
valuePerCriterionPerAlternative.entrySet()) {
            Alternative alternative = entry0.getKey();
            u.put(alternative, new HashMap<Criterion, double[]>());
            for (Map.Entry<Criterion, Double> entry1 :
entry0.getValue().entrySet()) {
                Criterion criterion = entry1.getKey();
                Double value = entry1.getValue();
                double[] steps = get(stepsPerCriterion, criterion);
                double[] zeros = new
double[criterion.getNumberOfDiscreteValues()];
                for (int i = 0; i < zeros.length; i++) {
                    zeros[i] = 0.0;
                }

                get(u, alternative).put(criterion, zeros);
                if (criterion.getAsceding()) {
                    for (int stepIndex = 0; stepIndex < (steps.length - 1);
stepIndex++) {
                        if (value <= steps[stepIndex + 1]) {
                            get(get(u, alternative), criterion)[stepIndex] = 1 -
((value - steps[stepIndex]) / (steps[stepIndex + 1] - steps[stepIndex]));
                            get(get(u, alternative), criterion)[stepIndex + 1] =
((value - steps[stepIndex]) / (steps[stepIndex + 1] - steps[stepIndex]));
                            break;

                        }
                    }
                } else {
                    for (int stepIndex = 0; stepIndex < (steps.length - 1);
stepIndex++) {
                        if (value >= steps[stepIndex + 1]) {
                            get(get(u, alternative), criterion)[stepIndex] = 1 -
((value - steps[stepIndex]) / (steps[stepIndex + 1] - steps[stepIndex]));
                            get(get(u, alternative), criterion)[stepIndex + 1] =
((value - steps[stepIndex]) / (steps[stepIndex + 1] - steps[stepIndex]));
                            break;

                        }
                    }
                }
            }
        }
        return u;
    }

    private static Map<Criterion, double[]>
retrieveStepsPerCriterion(Map<Alternative, Map<Criterion, Double>>
valuePerCriterionPerAlternative, List<Criterion> criteria) {
        Map<Criterion, double[]> toReturn = new HashMap<Criterion, double[]>();
        for (Criterion criterion : criteria) {
            int g = 0;
            double gd;
            double gu;
            if (criterion.getAsceding()) {
```

```
                gd =
Collections.min(retrieveCriterionValues(valuePerCriterionPerAlternative,
criterion));
                gu =

Collections.max(retrieveCriterionValues(valuePerCriterionPerAlternative,
criterion));
            } else {
                gd =
Collections.max(retrieveCriterionValues(valuePerCriterionPerAlternative,
criterion));
                gu =

Collections.min(retrieveCriterionValues(valuePerCriterionPerAlternative,
criterion));
            }

            double[] steps = new double[criterion.getNumberOfDiscreteValues()];
            for (int i = 0; i < steps.length; i++) {
                steps[i] = gd + i * (gu - gd) / (steps.length - 1);
            }

            toReturn.put(criterion, steps);
        }

        return toReturn;
    }

    /**
     *
     * @param valuePerCriterionPerAlternative
     * @param criterion
     * @return
     */
    private static List<Double> retrieveCriterionValues(Map<Alternative,
Map<Criterion, Double>> valuePerCriterionPerAlternative, Criterion criterion) {
        LOGGER.trace("retrieveCriterionValues().. criterion=" + criterion + "\t"
+ valuePerCriterionPerAlternative);
        List<Double> values = new ArrayList<Double>();
        for (Map<Criterion, Double> valuePerCriterion :
valuePerCriterionPerAlternative.values()) {
            //Επειδή το κλειδι μπορει να ειναι διαφορετικο instance απο της
παραμετρο criterion κανουμε ένα loop μέχρι να βρουμε το πως συνδέονται
            for (Map.Entry<Criterion, Double> entry :
valuePerCriterion.entrySet()) {
                if (entry.getKey().getId().equals(criterion.getId())) {
                    LOGGER.trace("retrieveCriterionValues().. add value=" +
entry.getValue());
                    values.add(entry.getValue());
                    break;
                }
            }
        }
        if (values.isEmpty()) {
            LOGGER.trace("retrieveCriterionValues().. no values!");
        }
        return values;


    }

    private static class AlternativeWithRank
            implements Comparable<AlternativeWithRank> {

        private Alternative alternative;
```

```
        private int rank;

        public AlternativeWithRank(Alternative alternative, int rank) {
            this.alternative = alternative;
            this.rank = rank;
        }

        public int compareTo(AlternativeWithRank o) {
            return o.rank == rank ?
(alternative.getId().compareTo(o.alternative.getId())) : rank - o.rank;
        }
    }

    private static <K extends AbstractEntity, V extends Object> V get(Map<K, V>
map, K key) {
        if (map == null) {
            return null;

        }
        for (Map.Entry<K, V> entry : map.entrySet()) {
            if (entry.getKey().getId().equals(key.getId())) {
                return entry.getValue();
            }
        }
        return null;
    }

    private static String toString(double[] values) {
        StringBuilder v = new StringBuilder();
        for (double vv : values) {
            v.append(vv).append(",");
        }
        return v.toString();
    }
}
```

NAI Algorithm Implementation (class com.gorbas.negotiation.NAI)

```
package com.gorbas.negotiation;

import com.gorbas.model.Alternative;
import com.gorbas.model.Contest;
import com.gorbas.model.DecisionMaker;
import com.gorbas.model.DecisionMakerAlternativeUsage;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collection;
import java.util.Collections;
import java.util.Comparator;
import java.util.HashMap;
import java.util.HashSet;
import java.util.List;
import java.util.Map;
import java.util.Set;

/**
 * Negotiable Alternatives Identifier for Negotiation Support (NAI)
implementation.
 * @author gorbas
 */
public class NAI {
	static final int MIN = 0;
	static final int MAX = 1;

	/**
	 * The classification of an alternative according to the preference set it
ended up into.
	 */
	public enum PrefClass {
		MostPreferred,
		Preferred,
		LeastPreferred
	}

	static class Proc2Data {
		public double[] t;
		public int[] i;
	}

	/**
	 * Groups metrics that refer to a specific alternative.
	 */
	public static class DmAltData {
		public double structuralIndex;
		public double preferenceRatio;
		public int rank;
		public DmAltData(double si, double pr, int r) {
			this.structuralIndex = si;
			this.preferenceRatio = pr;
			this.rank = r;
		}
	}

	/**
	 * Encapsulates the information and code of the NAI algorithm for a single
decision maker.
	 */
	static class DecisionMakerContext {
		private DecisionMaker dm;
		private DecisionMakerAlternativeUsage[] usages;
```

```
		private double[] rd; // Normalized preference values, may be smaller
than prefs, if running the algorithm on a subset of prefs.
		private double[] rdPartialSums;
		private double[] structuralIndices;
		private Proc2Data proc2Si;
		private double[] cd; // The preference ratios for all cut-off points
in the preferred subset.
		private Proc2Data proc2Cd;
		private int preferredCutOff = -1; // n*, the cut-off point
calculated by the expansion operation (separates the preferred from the least
preferred subset).
		private int mostPreferredCutOff = -1; // i*, the cut-off point
calculated by the contraction operation (separates the most-preferred from the
preferred subset).

		public DecisionMaker getDecisionMaker () { return dm; }

		DecisionMakerContext(DecisionMaker decisionMaker, List<Alternative>
alternatives) {
			assert(alternatives != null && decisionMaker != null);
			this.dm = decisionMaker;

			// Collect the utility values assigned by the decision maker
for all alternatives.
			usages = new
DecisionMakerAlternativeUsage[alternatives.size()];
			int i = 0;
			for (Alternative a : alternatives) {
				usages[i] =
DecisionMakerAlternativeUsage.retrieve(decisionMaker, a);
				if (usages[i] == null) {
					// Create a bogus usage value for any
alternative that has not been evaluated by the user.
					DecisionMakerAlternativeUsage dmau = new
DecisionMakerAlternativeUsage(a, 0);
					dmau.setDecisionMaker(decisionMaker);
					usages[i] = dmau;
				}
				i++;
			}

			// Now sort the alternatives by descending usage value.
			Arrays.sort(usages, new
Comparator<DecisionMakerAlternativeUsage>() {
				@Override
				public int compare(DecisionMakerAlternativeUsage o1,
DecisionMakerAlternativeUsage o2) {
					double diff = o1.getUsageValue() -
o2.getUsageValue();
					if (diff < 0) return 1;
					else if (diff == 0) return 0;
					else return -1;
				}
			});

			// Validate the usage values: Make sure no negative values
exist and that the sum is not zero.
			double sum = 0;
			for (DecisionMakerAlternativeUsage dmau : usages) {
				double u = dmau.getUsageValue();
				if (u < 0)
					throw new IllegalArgumentException("NAI:
usages[Alternative.Id=" + dmau.getAlternative().getId() + "] = " + u + ": utility
values must be non-negative.");
				sum += u;
```

```
                }
                if (sum <= 0)
                        throw new IllegalArgumentException("NAI: sum of
utility values must be positive.");
        }

        /**
         * Runs the algorithm on the specified number of alternatives,
         * taken off the top (most preferred) of the sorted alternative
list.
         * @param nrAlternatives The number of alternatives to consider in
the algorithm.
         *        These alternatives will be be the most preferred among
all.
         *        Non-positive and out-of-bounds values mean "consider all
alternatives".
         */
        public void run(int nrAlternatives) {
                init(nrAlternatives);
                // Do the expansion and contraction steps.
                doExpansion();
                doContraction();
        }

        public void init(int nrAlternatives) {
                reset();

                if (nrAlternatives <= 0 || nrAlternatives > usages.length)
                        nrAlternatives = usages.length;

                // Initialize the normalized preference values.
                if (null == this.rd || this.rd.length != nrAlternatives) {
                        double[] nu = new double[nrAlternatives];
                        for (int i = 0; i < nrAlternatives; i++) {
                                DecisionMakerAlternativeUsage dmau =
usages[i];
                                nu[i] = dmau.getUsageValue();
                             }
                        normalize(nu);
                        this.rd = nu;
                }
        }

        private void reset() {
                this.rd = null;
                this.rdPartialSums = null;
                this.structuralIndices = null;
                this.proc2Si = null;
                this.cd = null;
                this.proc2Cd = null;
                this.preferredCutOff = -1;
                this.mostPreferredCutOff = -1;
        }

        /**
         * Calculates and returns the partial sums of the i most preferred
alternatives, with i = 1..N, N the number of alternatives.
         * @return An array, Rd, with the sums. Rd[i] is the sum of the i+i
most-preferred alternatives, (rd[0] + ... + rd[i]).
         */
        private double[] getRdPartialSums() {
                if (null == rdPartialSums) {
                        rdPartialSums = new double[rd.length];
                        double sum = 0;
                        for (int i = 0; i < rd.length; i++)
```

```
                                rdPartialSums[i] = (sum += rd[i]);
                        }
                        return rdPartialSums;
                }

                /**
                 * Calculates the Structural Indices of all most-preferred
alternative subsets with size of 2 or greater.
                 * @return An array with the Structural Indices. Position i of the
array will contain the S.I. of the i+1 most-preferred alternatives.<br/>
                 *         The first position of the array will always contain 0
(S.I. is not defined for the subset that contains only the most preferred
alternative).
                 */
                private double[] getStructuralIndices() {
                        if (null == structuralIndices) {
                                double[] Rd = getRdPartialSums();
                                structuralIndices = new double[Rd.length];

                                for (int j = 1; j < structuralIndices.length; j++) {
                                        double Mdkj_sum = 0;
                                        for (int k = 0; k < j; k++) {
                                                // Optimization: Do not sum anymore if
already reached infinity.
                                                if (Mdkj_sum == Double.POSITIVE_INFINITY)
                                                        break;

                                                double Mdkj = (Rd[k] / (k+1)) / ((Rd[j] -
Rd[k]) / (j - k)); // "+1" because k is zero-based index.
                                                Mdkj_sum += Mdkj;
                                        }
                                        structuralIndices[j] =  Mdkj_sum / (j * (j+1));
                                }
                        }
                        return structuralIndices;
                }

                /**
                 * Performs the Expansion Operation of NAI and returns the cut-off
point (n*), which is the number of elements in the Preferred subset.
                 * @return The number of elements in the Preferred subset.
                 */
                private int doExpansion() {
                        if (preferredCutOff < 0) {
                                // If we have only one alternative, make that
preferred (Expansion needs at least 2 alternatives).
                                if (rd.length <= 1)
                                        preferredCutOff = rd.length;
                                else { // Perform Expansion.
                                        double siMin = Double.POSITIVE_INFINITY;
                                        int iMin = 0;
                                        double[] sil = getStructuralIndices();
                                        for (int i = 1; i < rd.length; i++) {
                                                if (siMin > sil[i]) {
                                                        siMin = sil[i];
                                                        iMin = i;
                                                }
                                        }
                                        preferredCutOff = iMin + 1;
                                }
                        }
                        return preferredCutOff;
                }

                /**
```

```
		 * Calculates the preference ratios.<br/>
		 * Assumes that Expansion has already run.
		 * @return An array with the preference ratios (array[i] contains
Cd(i+1)).
		 */
		private double[] getPreferenceRatios() {
			assert(preferredCutOff >= 0);
			if (null == cd) {
				cd = new double[preferredCutOff - 1];
				double[] prefNorm = Arrays.copyOfRange(rd, 0,
preferredCutOff);
				normalize(prefNorm);
				// Calculate and store the ratios.
				double sum = 0;
				for (int i = prefNorm.length - 1, j = 1; i > 0; i--,
j++) { // j = count of items in the bottom (preferred) subset.
					sum += prefNorm[i];
					double cdv = (prefNorm[i-1] * j) / sum;
					cd[i-1] = cdv;
				}
			}
			return cd;
		}

		/**
		 * Explicitly sets the preferred set cut-off point and makes sure
that contraction will use that point.
		 * @param prefCutOff
		 */
		private void prepareContraction(int prefCutOff) {
			preferredCutOff = prefCutOff;
			this.cd = null;
			mostPreferredCutOff = -1;
		}

		/**
		 * Performs the Contraction Operation of NAI and returns the cut-off
point (i*).<br/>
		 * Assumes that the Expansion Operation has already been run.
		 * @return The number of elements in the Most Preferred subset.
		 */
		private int doContraction() {
			assert(preferredCutOff >= 0);
			if (mostPreferredCutOff < 0) {
				// If we have only 1 alternative, make that the most
preferred one.
				if (preferredCutOff <= 1)
					mostPreferredCutOff = preferredCutOff;
				else {
					double[] pr = getPreferenceRatios();
					// Find-out the maximum Preference Ratio that
determines the cut-off point for the most-preferred subset.
					double max = Double.NEGATIVE_INFINITY;
					int iMax = 0;
					boolean indifferent = true;
					for (int i = 0; i < pr.length; i++) {
						if (pr[i] >= max) {
							max = pr[i];
							iMax = i;
						}
						if (indifferent && i > 0 && pr[i] !=
pr[i-1])
							indifferent = false;
					}
```

```
                                // If all preferences are indifferent, set the
cut-off point to the maximum.
                                mostPreferredCutOff = indifferent ?
preferredCutOff : iMax + 1;
                          }
                    }
                    return mostPreferredCutOff;
             }

             /**
              * @return The size of the most-preferred set of alternatives.
              */
             public int getMostPreferred() {
                    doExpansion();
                    return doContraction();
             }

             /**
              * @return The size of the preferred set of alternatives (includes
the most-preferred set).
              */
             public int getPreferred() {
                    return doExpansion();
             }

             public PrefClass getAlternativeClassification(Alternative a) {
                    int i = 0;
                    for (; i < usages.length; i++) {
                          if
(usages[i].getAlternative().getId().equals(a.getId())) {
                                break;
                          }
                    }
                    if (i >= preferredCutOff)
                          return PrefClass.LeastPreferred;
                    if (i >= mostPreferredCutOff)
                          return PrefClass.Preferred;
                    return PrefClass.MostPreferred;
             }

             /**
              * Returns the specified amount of most preferred alternatives.
              * @param count The number of alternatives to return.
              * @return A list with the alternatives.
              */
             public List<Alternative> getTopAlternatives(int count) {
                    if (count < 0) count = 0;
                    if (count > usages.length) count = usages.length;
                    List<Alternative> top = new ArrayList<Alternative>(count);
                    for(int i = 0; i < count; i++)
                          top.add(usages[i].getAlternative());
                    return top;
             }

             /**
              * Exports the data calculated by the algorithm.
              * @return
              */
             public Map<Alternative, DmAltData> export() {
                    HashMap<Alternative, DmAltData> m = new HashMap<Alternative,
DmAltData>();
                    for (int i = 0; i < usages.length; i++) {
                          DecisionMakerAlternativeUsage u = usages[i];
                          Alternative a = u.getAlternative();
```

```
                        DmAltData d = new DmAltData(Double.NaN, Double.NaN, i
+ 1);
                        if (null != structuralIndices && i <
structuralIndices.length)
                              d.structuralIndex = structuralIndices[i];
                        if (null != cd && i < cd.length)
                              d.preferenceRatio = cd[i];
                        m.put(a, d);
                  }
                  return m;
            }

            public void calculateProc2Step12Data() {
                  if (mostPreferredCutOff > 0) { // Contraction must have run
before this.
                        proc2Cd = new Proc2Data();
                        if (cd.length > mostPreferredCutOff) {
                              double[] bounds = getBounds(cd,
mostPreferredCutOff, cd.length);
                              double span = bounds[MAX] - bounds[MIN];
                              HashMap<Double, Integer> t2i = new
HashMap<Double, Integer>();
                              // Calculate all t that satisfy cd[i] = max -
t*(max - min).
                              for (int i = mostPreferredCutOff; i < cd.length;
i++) {
                                    if (span > 0)
                                          t2i.put((bounds[MAX] - cd[i]) /
span, i);
                                    else
                                          t2i.put(0.0, i);
                              }
                              // Now sort with ascending t.
                              proc2Cd.t = new double[t2i.size()];
                              proc2Cd.i = new int[t2i.size()];
                              int i = 0;
                              for (Double t : t2i.keySet())
                                    proc2Cd.t[i++] = t;
                              Arrays.sort(proc2Cd.t);
                              // Construct the array of indices that
correspond to the threshold values.
                              for (i = 0; i < proc2Cd.t.length; i++)
                                    proc2Cd.i[i] = t2i.get(proc2Cd.t[i]);
                        }
                        else {
                              proc2Cd.t = new double[0];
                              proc2Cd.i = new int[0];
                        }
                  }
            }

            public void calculateProc2Step34Data() {
                  if (preferredCutOff >= 0) { // Expansion must have run
before.
                        proc2Si = new Proc2Data();
                        if (structuralIndices.length > preferredCutOff) {
                              double[] bounds = getBounds(structuralIndices,
preferredCutOff, structuralIndices.length);
                              double span = bounds[MAX] - bounds[MIN];
                              HashMap<Double, Integer> t2i = new
HashMap<Double, Integer>();
                              // Calculate all t that satisfy SI[i] = min +
t*(max - min).
                              for (int i = preferredCutOff; i <
structuralIndices.length; i++) {
```

```
                                        if (span > 0)
                                              t2i.put((structuralIndices[i] -
bounds[MIN]) / span, i);
                                        else
                                              t2i.put(0.0, i);
                                  }
                                  // Now sort with ascending t.
                                  proc2Si.t = new double[t2i.size()];
                                  proc2Si.i = new int[t2i.size()];
                                  int i = 0;
                                  for (Double t : t2i.keySet())
                                        proc2Si.t[i++] = t;
                                  Arrays.sort(proc2Si.t);
                                  // Construct the array of indices that
correspond to the threshold values.
                                  for (i = 0; i < proc2Si.t.length; i++)
                                        proc2Si.i[i] = t2i.get(proc2Si.t[i]);
                            }
                            else {
                                  proc2Si.t = new double[0];
                                  proc2Si.i = new int[0];
                            }
                      }
                }
        }

        private Contest contest;
        private List<Alternative> alternatives;
        private HashMap<DecisionMaker, DecisionMakerContext> decisionMakers;
        private List<DecisionMaker> dmList;

        public NAI(Contest contest) { this(contest, null); }

        /**
         * Creates a NAI instance.
         * @param contest
         * @param alternatives The list of alternatives to consider in this
algorithm.
         *        By default, this will be the list of alternatives of the contest.
         */
        public NAI(Contest contest, List<Alternative> alternatives) {
                if (null == alternatives || alternatives.isEmpty())
                      alternatives = contest.getAlternativeList();
                List<DecisionMaker> dml = contest.getDecisionMakerList();
                HashMap<DecisionMaker, DecisionMakerContext> dmcl = new
HashMap<DecisionMaker, DecisionMakerContext>();
                int i = 0;
                for (DecisionMaker dm : dml)
                      dmcl.put(dm, new DecisionMakerContext(dm, alternatives));
                this.decisionMakers = dmcl;
                this.contest = contest;
                this.alternatives = alternatives;
                this.dmList = dml;
        }

        public Contest getContest() { return contest; }

        public List<Alternative> getAlternatives() { return
Collections.unmodifiableList(alternatives); }

        public List<DecisionMaker> getDecisionMakers() { return
Collections.unmodifiableList(dmList); }

        /**
```

```
	 * Returns the kind of the set into which an alternative has been
classified for the specified decision maker.
	 * @param dm
	 * @param a
	 * @return
	 */
	public PrefClass getClassification(DecisionMaker dm, Alternative a) {
		DecisionMakerContext ctx = decisionMakers.get(dm);
		assert(null != ctx);
		if (null == ctx) return PrefClass.LeastPreferred;
		return ctx.getAlternativeClassification(a);
	}

	/**
	 * Exports the algorithm state.
	 * @return
	 */
	public Map<DecisionMaker, Map<Alternative, DmAltData>> export() {
		Map<DecisionMaker, Map<Alternative, DmAltData>> m = new
HashMap<DecisionMaker, Map<Alternative, DmAltData>>();
		for(Map.Entry<DecisionMaker, DecisionMakerContext> e :
decisionMakers.entrySet())
			m.put(e.getKey(), e.getValue().export());
		return m;
	}

	/**
	 * Resets the algorithm state.
	 */
	private void reset() {
		for (DecisionMakerContext ctx : decisionMakers.values())
			ctx.reset();
	}

	/**
	 * Runs the NAI algorithm.<br/>
	 * If the intersection of the most-preferred sets is empty after the run,
"Procedure 1" is attempted.<br/>
	 * If the intersection is still empty, then "Procedure 2" is applied.
	 */
	public void run() {
		reset();
		dmRun();

		int altSize = alternatives.size();
		if (altSize <= 1) return;

		Set<Alternative> common = getCommonMostPreferred();
		if (!common.isEmpty())
			return;

		// Apply procedure 1 until we either get a non-empty intersection or
we cannot apply it further.
		while (common.isEmpty() && procedure1())
			common = getCommonMostPreferred();
		if (!common.isEmpty())
			return;

		// Apply procedure 2 to get a non-empty intersection.
		reset();
		dmRun();
		procedure2();
	}

	/**
```

```
	 * Runs the basic step of the algorithm on all decision makers.
	 */
	private void dmRun() {
		for (DecisionMakerContext ctx : decisionMakers.values())
			ctx.run(-1);
	}

	/**
	 * Implements an iteration of "Procedure 1", which trims away the least-
preferred alternatives. This is done in hope that
	 * the next run of NAI over the trimmed alternative sets will yield a non-
empty common most-preferred set.
	 * @return False if there is no need to run another iteration (if there is
no more to trim from the alternative set).
	 */
	private boolean procedure1() {
		boolean more = false;
		for (DecisionMakerContext ctx : decisionMakers.values()) {
			assert (ctx.preferredCutOff > -1);
			int co = ctx.preferredCutOff;
			ctx.run(ctx.preferredCutOff);
			if (ctx.preferredCutOff != co)
				more = true;
		}
		return more;
	}

	/**
	 * Implements "Procedure 2".
	 */
	private void procedure2() {
		// If the preferred sets of all decision makers have any common
items then we can apply steps 1 and 2,
		// otherwise we need to apply steps 3 and 4 first.
		Set<Alternative> comPref = getCommonPreferred();
		double[] thresholds;
		if (comPref.isEmpty()) { // Apply steps 3 and 4.
			thresholds = null; // This will hold all the values of "t"
that are worth trying.
			for(DecisionMakerContext dm : decisionMakers.values()) {
				dm.calculateProc2Step34Data();
				thresholds = merge(thresholds, dm.proc2Si.t);
			}
			// For each threshold value "t"...
			for(double t : thresholds) {
				// Try expanding the preferred set of each decision
maker until we get any item in their intersection.
				for (DecisionMakerContext dm :
decisionMakers.values()) {
					// The upper structural index for this decision
maker would be:
					//   siUpper = siMin + t * (siMax - siMin);
					// We need to find the item with the maximum SI
below or at this limit, which is equivalent to locating the
					// item with the "t" value that is nearest from
below to the current "t" value.
					double[] tvals = dm.proc2Si.t;
					int[] indices = dm.proc2Si.i;
					int idx = 0;
					for (int i = 0; i < tvals.length && t >=
tvals[i]; i++)
						idx = indices[i];
					// Now use the item index as the new preferred
set cut-off point.
					dm.prepareContraction(idx);
```

```
                                dm.doContraction();
                            }
                            // Find the intersection
                            comPref = getCommonPreferred();
                            if (!comPref.isEmpty())
                                break; // We found the minimum t value that
allows at least one item in the intersection.
                        }
                }

                // Apply steps 1 and 2 (extend the most-preferred sets until we find
common items in their intersection).
                comPref = getCommonMostPreferred();
                if (comPref.isEmpty()) {
                        thresholds = null;
                        for (DecisionMakerContext dm : decisionMakers.values()) {
                                dm.calculateProc2Step12Data();
                                thresholds = merge(thresholds, dm.proc2Cd.t);
                        }
                        // For each threshold value "t"...
                        for (double t : thresholds) {
                                // Try expanding the most-preferred set of each
decision maker until we get any item in their intersection.
                            for (DecisionMakerContext dm :
decisionMakers.values()) {
                                // The lower preference ratio for this decision
maker would be:
                                //   cdLower = cdMin - t * (cdMax - cdMin);
                                // We need to find the item with the minimum Cd
above or at this limit, which is equivalent to locating the
                                // item with the "t" value that is nearest from
below to the current "t" value.
                                double[] tvals = dm.proc2Cd.t;
                                int[] indices = dm.proc2Cd.i;
                                int idx = 0;
                                for (int i = 0; i < tvals.length && t >=
tvals[i]; i++)
                                        idx = indices[i];
                                // Now use the item index as the new preferred
set cut-off point.
                                dm.mostPreferredCutOff = idx;
                            }
                            // Find the intersection
                            comPref = getCommonMostPreferred();
                            if (!comPref.isEmpty())
                                break; // We found the minimum t value that
allows at least one item in the intersection.
                        }
                }
        }

        public List<Alternative> getMostPreferredOf(DecisionMaker dm) {
                DecisionMakerContext ctx = decisionMakers.get(dm);
                if (null != ctx)
                        return ctx.getTopAlternatives(ctx.getMostPreferred());
                return Collections.emptyList();
        }

        public List<Alternative> getPreferredOf(DecisionMaker dm) {
                DecisionMakerContext ctx = decisionMakers.get(dm);
                if (null != ctx)
                        return ctx.getTopAlternatives(ctx.getPreferred());
                return Collections.emptyList();
        }
```

```
	/**
	 * Returns the intersection of the most-preferred sets of all decision
makers.
	 * @return A set with the common alternatives.
	 */
	public Set<Alternative> getCommonMostPreferred() {
		HashSet<Alternative> common = new HashSet<Alternative>();
		boolean init = true;
		for (DecisionMakerContext ctx : decisionMakers.values()) {
			List<Alternative> prefs =
ctx.getTopAlternatives(ctx.getMostPreferred());
			if (init) {
				common.addAll(prefs);
				init = false;
			}
			else common.retainAll(prefs);
		}
		return common;
	}

	/**
	 * Returns the intersection of the preferred sets of all decision makers.
	 * @return A Set with the common alternatives.
	 */
	public Set<Alternative> getCommonPreferred() {
		HashSet<Alternative> common = new HashSet<Alternative>();
		boolean init = true;
		for (DecisionMakerContext ctx : decisionMakers.values()) {
			List<Alternative> prefs =
ctx.getTopAlternatives(ctx.getPreferred());
			if (init) {
				common.addAll(prefs);
				init = false;
			}
			else common.retainAll(prefs);
		}
		return common;
	}

	private static void normalize(double[] array) {
		double sum = 0;
		for (int i = 0; i < array.length; i++)
			sum += array[i];
		assert(sum > 0);

		for (int i = 0; i < array.length; i++)
			array[i] /= sum;
	}

	/**
	 * Finds the minimum and maximum in an array of numbers.
	 * @param array
	 * @param start
	 * @param end
	 * @return An array, {minimum, maximum}.
	 */
	private static double[] getBounds(double[] array, int start, int end) {
		double min, max;
		if (start >= end) {
			min = max = 0;
		}
		else {
			min = Double.POSITIVE_INFINITY;
			max = Double.NEGATIVE_INFINITY;
```

```
                for (int i = start; i < end; i++) {
                        if (array[i] < min) min = array[i];
                        if (array[i] > max) max = array[i];
                }
        }
        return new double[] {min, max};
    }

    private static double[] merge(double[] a1, double[] a2) {
        if (a1 == null && a2 == null) return new double[0];
        if (a1 == null) return a2;
        if (a2 == null) return a1;

        double[] m = new double[a1.length + a2.length];
        int i = 0;
        int j = 0;
        int k = 0;
        while (i < a1.length && j < a2.length) {
                if (a1[i] < a2[j] || Double.isNaN(a1[i]))
                        m[k] = a1[i++];
                else if (a1[i] == a2[j]) {
                        m[k] = a1[i++];
                        j++;
                }
                else {
                        m[k] = a2[j++];
                }
                k++;
        }
        while (i < a1.length)
                m[k++] = a1[i++];
        while (j < a2.length)
                m[k++] = a2[j++];
        if (k < m.length)
                m = Arrays.copyOfRange(m, 0, k);
        return m;
    }
}
```

Argumentation Protocol (class com.gorbas.negotiation.amgoud.*)

act.java

```
package com.gorbas.negotiation.amgoud;

public enum Act {
	Offer,
	Challenge,
	Argue,
	Withdraw,
	Accept,
	Refuse,
	SayNothing;

	/**
	 *
	 * @param Returns the possible response acts for a received act.
	 * @return The possible response acts, according to the protocol.
	 */
	public static Act[] getReplies(Act a) {
		if (a == Offer) return new Act[] {Accept, Refuse, Challenge};
		if (a == Challenge) return new Act[] {Argue};
		if (a == Argue) return new Act[] {Accept, Challenge, Argue};
		if (a == Accept || a == Refuse) return new Act[] {Accept, Challenge,
Argue, Withdraw};
		// a == WithDraw
		return new Act[0];
	}
}
```

agent.java

```
package com.gorbas.negotiation.amgoud;

import java.util.*;

/**
 * Represents a negotiation partaker.
 * @author gorbas
 */
public abstract class Agent {
	protected Protocol negProtocol;
	private LinkedList<Move> inbox = new LinkedList<Move>(); // The incoming
moves that need replies.
	private boolean initiator;

	// Commitment store.

	/**
	 * The offers accepted or proposed by this agent.
	 */
	protected Set<Offer> acceptedOffers = new HashSet<Offer>();
	protected Set<Offer> rejectedOffers = new HashSet<Offer>();

	protected Set<Argument> usedArguments = new HashSet<Argument>();
	protected Set<Move.Challengable> usedChallenges = new
HashSet<Move.Challengable>();
	protected Set<Argument> acquiredBeliefs = new HashSet<Argument>(); // The
acceptable arguments we added to the agent's knowledge base.


	/**
	 * Creates an agent.
	 * @param p The negotiation protocol.
	 */
	public Agent(Protocol p) {
		this.negProtocol = p;
	}

	/**
	 * Returns the accepted offers. We expect only one offer at most per round.
	 * @return
	 */
	public Set<Offer> getAcceptedOffers() { return
Collections.unmodifiableSet(acceptedOffers); }

	/**
	 * Returns all arguments in favor of an offer.
	 * @param a The agent whose goals are evaluated.
	 *   If it is this agent instance, arguments that satisfy goals in Gi will
be returned.
	 *   For other agents, GOi,j will be used. (i = this agent, j = the other
agents).
	 * @param o The offer to argue about.
	 * @param against False: Arguments in favor of offer will be returned.<br/>
	 *  True: Arguments against the offer will be returned.
	 * @return The list of all arguments for the offer.
	 */
	protected abstract List<? extends Argument> getArgumentsForOffer(Agent a,
Offer o, boolean against);

	/**
	 * Tries to create an acceptable argument for a move.
	 * @param challenger The agent that we would like to persuade about our
move.
```

```
         * @param m The move that was challenged.
         * @param against True if arguments against the move are required.
         * @return The argument or null if an acceptable argument cannot be found.
         */
        protected abstract Argument findAcceptableArgument(Agent challenger,
Move.Challengable m, boolean against);

        /**
         * Decides whether an argument is acceptable by this agent.
         * @param a
         * @return
         */
        protected abstract boolean isAcceptable(Argument a);

        /**
         * Detects strong (weakness = 0) CON arguments against an offer.
         * @param o
         * @return True if there is at least one strong argument against an offer.
         */
        private boolean existsStrongArgumentAgainst(Offer o) {
                for (Agent a : negProtocol.getAgents()) {
                        for (Argument arg : getArgumentsForOffer(a, o, true)) {
                                if (arg.getWeakness() == 0)
                                        return true;
                        }
                }
                return false;
        }

        /**
         * Chooses the non-discussed offer with the strongest arguments with
respect to the goals of this agent,
         *  and without strong arguments (weakness = 0)
         *  against the offer with respect to the goals of the other agents.
         * @return The best offer or null, if none satisfies the criteria.
         */
        private Offer pickGoodOffer() {
                Argument a = null;
                Offer mostPreferred = null;
                for (Offer o : negProtocol.getOffers()) {
                        if (existsStrongArgumentAgainst(o))
                                continue;
                        // Choose the offer with the most preferred (most powerful)
PRO argument in favor of the offer.
                        if (null == mostPreferred || negProtocol.isAPreferredOverB(o,
mostPreferred, this, false))
                                mostPreferred = o;
                }
                return mostPreferred;
        }

        private boolean isAcceptable(Offer x) {
                List<? extends Offer> offers = negProtocol.getOffers();
                if (!offers.contains(x))
                        return false;
                // Find the most preferred offer (has the strongest argument in
favor of it).
                Offer mp = x;
                Argument mpa = null;
                for (Offer o : offers) {
                        if (negProtocol.isAPreferredOverB(o, mp, this, false))
                                mp = o;
                }
                return x.equals(mp);
        }
```

```
	/*
	 * Notifications from the protocol manager.
	 */

	void clearCommitmentStore() {
		acceptedOffers.clear();
		rejectedOffers.clear();
		usedArguments.clear();
		usedChallenges.clear();
	}

	void dialogueEnded() {
		inbox.clear();
	}

	void sendMove(Move incoming) {
		this.inbox.add(incoming);
		Agent speaker = incoming.getSpeaker();
		switch (incoming.getAct()) {
			case Argue: // Postcondition: Accept all acceptable
arguments.
				{
					Argument a =
((Move.Argue)incoming).getArgument();
					if (!acquiredBeliefs.contains(a) &&
isAcceptable(a))
						acquiredBeliefs.add(a);
				}
				break;
		}
	}

	/**
	 * @param offer
	 * @return True if the offer has been rejected by this agent.
	 */
	public boolean hasRejected(Offer offer) {
		return rejectedOffers.contains(offer);
	}

	/**
	 * @param offer
	 * @return True if the offer has been accepted by this agent.
	 */
	public boolean hasAccepted(Offer offer) {
		return acceptedOffers.contains(offer);
	}

	/*
	 * Agent actions.
	 */

	/**
	 * Attempts to offer the most-preferred, non-discussed alternative.
	 */
	private void offer() {
		Offer mp = pickGoodOffer();
		if (null == mp) // No appropriate offer found, just withdraw.
			negProtocol.withDraw(this);
		else {
			negProtocol.offer(this, mp);
			acceptedOffers.add(mp);
		}
	}
```

```
	/**
	 * Argues against another agent's move.
	 * Move m
	 * @return True if there was an argument.
	 */
	private boolean tryArgue(Move m) {
		Move.Challengable cm = null;
		Agent hearer = m.getSpeaker(); // We respond to the agent that
challenges us.
		Agent argumentSource; // The agent whose arguments we must consult.
		boolean against = false;
		Argument acceptable = null;
		switch(m.getAct()) {
			case Accept:
				cm =
((Move.Accept)m).getAcceptedMoves().iterator().next();
				if (this.equals(cm.getSpeaker()))
					return false; // We do not argue against sb that
accepted our move.
				against = true;
				argumentSource = hearer;
				break;
			case Refuse:
				cm = (Move.Refuse)m;
				against = true; // Against the refusal, for our offer.
				argumentSource = hearer;
				break;
			case Argue:
				cm = ((Move.Argue)m).getChallenged();
				Argument otherArg = ((Move.Argue)m).getArgument();
				against = otherArg.isPro() ^ cm.getAct() ==
Act.Refuse; // We need arguments of the opposite kind of the opponent argument.
				argumentSource = otherArg.getSubject();
				break;
			case Challenge:
				cm =
((Move.Challenge)m).getChallenged().iterator().next();
				if (!cm.getSpeaker().equals(this))
					return false; // Do not answer challenges for
others.
				against = false;
				argumentSource = this; // We need to defend our move.
				if (cm.getAct() == Act.Refuse) {
					// When defending a refusal we should give one
of the arguments we based the refusal on.
					List<? extends Argument> refArgs =
getArgumentsForOffer(this, ((Move.Refuse)cm).getOffer(), true);
					acceptable =
Argument.findMostPreferred(refArgs);
				}
				break;
			default: return false;
		}
		if (null == acceptable)
			acceptable = findAcceptableArgument(argumentSource, cm,
against);

		boolean argued = null != acceptable;
		if (argued) {
			negProtocol.argue(this, Collections.singletonList(hearer),
acceptable, cm);
			usedArguments.add(acceptable);
		}
		return argued;
```

```
	}

	/**
	 * Allows the Agent to initiate a dialog by making the first offer.
	 */
	public void becomeInitiator() {
		initiator = true;
	}

	/**
	 * Allows the Agent to perform its turn.
	 */
	public void doTurn() {
		// The first movement of each round (dialogue) is to make an offer.
		if (initiator) {
			initiator = false;
			offer();
			return;
		}

		boolean saidSomething = false;
		for (Move m : inbox) {
			// Try arguing in response to challenges first, otherwise try
to accept/refuse and finally, challenge.
			saidSomething |=
					tryArgue(m) ||
					tryAccept(m) ||
					tryRefuse(m) ||
					tryChallenge(m);
		}
		inbox.clear();

		if (!saidSomething) {
			// Did not respond. Say nothing.
			negProtocol.sayNothing(this);
		}
	}

	private void accept(Offer o) {
		acceptedOffers.add(o);
		rejectedOffers.remove(o);
	}

	private void reject(Offer o) {
		acceptedOffers.remove(o);
		rejectedOffers.add(o);
	}

	/**
	 * Attempts to accept an offer or argument.
	 * @param m The move to try accepting.
	 * @return true if accepted.
	 */
	private boolean tryAccept(Move m) {
		if (m instanceof Move.Challengable) { // Challengable moves are also
those that can be accepted.
			switch(m.getAct()) {
				case Offer: // Preconditions: The offer is the most
pessimistically preferred decision
					{
						Offer o = ((Move.OfferMove)m).getOffer();
						if (isAcceptable(o)) {
							if (acceptedOffers.contains(o))

	negProtocol.sayNothing(this); // Already accepted.
```

```
                                                else {
                                                       accept(o);
                                                       negProtocol.accept(this,
(Move.OfferMove)m);
                                                }
                                                return true;
                                         }
                                  }
                                  break;
                           case Argue: // Preconditions: The argument is
acceptable.
                                  {
                                         List<Move.Argue> accArgs = new
ArrayList<Move.Argue>();
                                         Argument a =
((Move.Argue)m).getArgument();
                                         if (isAcceptable(a)) {
                                                usedArguments.add(a);
                                                accArgs.add((Move.Argue)m);
                                                // Depending on the argument we
accept, implicitly accept or reject the offer.
                                                if
(((Move.Argue)m).getArgument().isCon())

       reject(((Move.Argue)m).getOffer());
                                                else

       accept(((Move.Argue)m).getOffer());
                                         }
                                         if (!accArgs.isEmpty()) {
                                                negProtocol.accept(this, accArgs);
                                                return true;
                                         }
                                  }
                                  break;
                           }
                    }
                    return false;
             }

             /**
              * Attempts to refuse an offer.
              * @param m The offer move to refuse.
              * @return true if there was an offer and this agent refused.
              */
             private boolean tryRefuse(Move m) {
                    if (m instanceof Move.OfferMove) {
                           Offer x = ((Move.OfferMove)m).getOffer();
                           // Precondition: There is at least one argument against the
offer.
                           List<? extends Argument> args = getArgumentsForOffer(this, x,
true);
                           if (args.isEmpty())
                                  return false;
                           // Refuse the offer.
                           reject(x);
                           negProtocol.refuse(this, x, m.getSpeaker());
                           return true;
                    }
                    return false;
             }

             /**
              * Decides whether a move should be challenged.
```

```
        * @param challenged The list of challenged moves. The moves that should
be challenged
        *  are added to this list.
        * @param m The move to check.
        */
       private void checkChallenge(Set<Move.Challengable> challenged,
Move.Challengable m) {
              if (usedChallenges.contains(m) || challenged.contains(m))
                     return; // Already challenged this one.
              if (m.getSpeaker().equals(this))
                     return; // Do not challenge own moves.
              switch (m.getAct()) {
                     case Offer:
                            {
                                   Offer x = ((Move.OfferMove)m).getOffer();
                                   // Precondition: There is another non-rejected
offer that is pessimistically
                                   // preferred to this one.
                                   for (Offer o : negProtocol.getOffers()) {
                                          if (!o.equals(x) &&
negProtocol.isAPreferredOverB(o, x, this, false)) {
                                                 challenged.add((Move.OfferMove)m);
                                                 break;
                                          }
                                   }
                            }
                            break;
                     // Other moves are challengable without preconditions.
                     case Argue:
                            {
                                   Move.Challengable mm =
((Move.Argue)m).getChallenged();
                                   checkChallenge(challenged, mm);
                            }
                            break;
                     case Accept:
                            {
                                   // See if there is any accepted move that
justifies challenging the accept.
                                   Set<Move.Challengable> acm = new
HashSet<Move.Challengable>();
                                   for (Move.Challengable mm :
((Move.Accept)m).getAcceptedMoves())
                                          checkChallenge(acm, mm);
                                   if (!acm.isEmpty()) // Should challenge the
accept.
                                          challenged.add((Move.Accept)m);
                            }
                            break;
                     case Refuse:
                            {
                                   // Challenge a refuse, but not if we have
already refused the same offer ourselves.
                                   if (!hasRejected(((Move.Refuse)m).getOffer()))
                                          challenged.add((Move.Challengable)m);
                            }
                            break;
              }
       }

       /**
        * Attempts to challenge the challengable incoming moves.
        * @param m The move to challenge.
        * @return True if this agent placed a challenge.
        */
```

```
	private boolean tryChallenge(Move m) {
		Set<Move.Challengable> chMoves = new HashSet<Move.Challengable>();
		if (m instanceof Move.Challengable)
			checkChallenge(chMoves, (Move.Challengable)m);

		if (!chMoves.isEmpty()) {
			usedChallenges.addAll(chMoves);
			negProtocol.challenge(this, chMoves);
			return true;
		}
		return false;
	}
}
```

argument.java

```
package com.gorbas.negotiation.amgoud;

import java.util.Collection;

/**
 * Represents a negotiation argument.
 * @author gorbas
 */
public abstract class Argument {
	/**
	 * Returns the agent to whom this argument refers.
	 * @return
	 */
	public abstract Agent getSubject();
	/**
	 * @return True if this is an argument in favor of some decision (PRO
argument).
	 */
	public abstract boolean isPro();

	/**
	 * @return True if this is an argument against one decision (CON argument).
	 */
	public final boolean isCon() { return !isPro(); }

	public abstract double getLevel();
	public abstract double getWeight();
	/**
	 * Returns m(v), with m being the order-reversing map of the scale of
value v.<br/>
	 * The default implementation is just m(v) = 1 - v, which satisfies the
conditions m(0) = 1, m(1) = 0 and v1 > v2 <=> m(v1) < m(v2).
	 * @param v The value to reverse, assumed to be in [0,1].
	 * @return The corresponding value to v in the reversed scale.
	 */
	protected double m(double v) { return 1.0 - v; }

	/**
	 * Gets the strength of the argument (used with PRO arguments).
	 * @return The argument strength (PRO argument) or the reversed argument
weakness (CON argument),
	 */
	public final double getStrength() {
		double l = getLevel(), w = getWeight();
		boolean pro = isPro();
		return pro ? Math.min(l, w) : m(Math.max(l, w));
	}

	/**
	 * Gets the weakness of the argument (used with CON arguments).
	 * @return The argument weakness (CON argument) or the reversed argument
strength (PRO argument),
	 */
	public final double getWeakness() {
		double l = getLevel(), w = getWeight();
		boolean pro = isPro();
		return pro ? m(Math.min(l, w)) : Math.max(l, w);
	}

	/**
	 * Decides whether this argument is preferred to an other argument.
	 * @param other
	 * @return True if this argument is the preferred one, false otherwise.
```

```
         */
        public final boolean isPreferredTo(Argument other) {
                boolean isPro = isPro();
                if (isPro() != other.isPro())
                        return false; // Cannot compare PRO with CON arguments.

                return isPro ? Math.min(getLevel(), getWeight()) >=
Math.min(other.getLevel(), other.getWeight())
                             : Math.max(getLevel(), getWeight()) >=
Math.max(other.getLevel(), other.getWeight());
        }

        /**
         * Finds the most preferred argument of a collection of arguments.
         * @param args The arguments.
         * @return The most-preferred argument or null if the collection was empty.
         */
        public static Argument findMostPreferred(Collection<? extends Argument>
args) {
                Argument mp = null;
                // Successively compare all arguments to the current most preferred
and update it if needed.
                for (Argument a : args) {
                        if (null == mp || a.isPreferredTo(mp))
                                mp = a;
                }
                return mp;
        }
}
```

dialogue.java

```
package com.gorbas.negotiation.amgoud;

import java.util.Collections;
import java.util.List;
import java.util.ArrayList;

/**
 * A finite sequence of moves of the negotiation protocol.
 * @author gorbas
 */
public class Dialogue {
	List<Move> moves = new ArrayList<Move>();
	Offer result;

	public Dialogue(List<Move> moves, Offer result) {
		this.moves = moves;
		this.result = result;
	}

	public List<Move> getMoves() { return Collections.unmodifiableList(moves);
}

	/**
	 * Returns the dialogue result.
	 * @return The offer all agents agreed upon. Null if the negotiation
failed.
	 */
	public Offer getResult() {
		return result;
	}

	/**
	 * @return True if the the participants have reached agreement.
	 */
	public boolean isSuccessful() {
		return result != null;
	}
}
```

move.java

```
package com.gorbas.negotiation.amgoud;

import java.util.*;

/**
 * Represents a move, which is a negotiation act by a speaker that is addressed
to one or more hearers.<br/>
 * The speaker cannot be a hearer.
 * @author gorbas
 */
public abstract class Move {
	protected Agent speaker;
	protected List<Agent> hearers;
	protected Act act;
	protected int id;

	public static Set<Agent> getSpeakers(Collection<? extends Move> ml) {
		HashSet<Agent> agents = new HashSet<Agent>();
		for (Move m : ml)
			agents.add(m.getSpeaker());
		return agents;
	}

	public Agent getSpeaker() {
		return speaker;
	}

	public List<Agent> getHearers() {
		return Collections.unmodifiableList(hearers);
	}

	public Act getAct() {
		return act;
	}

	public int getId() { return id; }

	@Override
	public boolean equals(Object obj) {
		return this == obj || (obj instanceof Move && ((Move)obj).id ==
this.id);
	}

	@Override
	public int hashCode() {
		return new Integer(id).hashCode();
	}

	protected Move(List<Agent> hearers, Agent speaker, Act act, int id) {
		this.hearers = hearers;
		this.speaker = speaker;
		this.act = act;
		this.id = id;
	}

	protected String strHeader() {
		return "#" + id + " Agent " + speaker + " to {" +
strAgentList(hearers) + "}: ";
	}

	protected String strBody() {
		return "Move " + id;
	}
```

```
	private static String strAgentList(Collection<? extends Agent> agents) {
		if (agents == null || agents.isEmpty())
			return "";
		StringBuilder sb = new StringBuilder();
		for (Agent a : agents)
			sb.append(a).append(", ");
		sb.delete(sb.length() - 2, sb.length());
		return sb.toString();
	}

	private static String strList(Collection<? extends Move> moves) {
		if (moves.size() > 1) {
			StringBuilder sb = new StringBuilder();
			sb.append("moves  ");
			for(Move m : moves)
				sb.append(m.id).append(", ");
			sb.delete(sb.length() - 2, sb.length());
			return sb.toString();
		}
		else if (moves.size() == 1) {
			Move m = moves.iterator().next();
			return "#" + m.id + ":" + m.strBody();
		}
		return "";
	}

	@Override
	public String toString() {
		return strHeader() + strBody();
	}

	public static class Challengable extends Move {
		protected Offer offer;
		protected boolean isAgainst;

		protected Challengable(List<Agent> hearers, Agent speaker, Act act,
Offer offer, int id) {
			super(hearers, speaker, act, id);
			this.offer = offer;
		}

		/**
		 * Returns the offer discussed in this move.
		 * @return
		 */
		public Offer getOffer() { return offer; }

		public boolean isAgainstOffer() {
			return isAgainst;
		}
	}

	/**
	 * Represents an offer.
	 */
	public static class OfferMove extends Challengable {
		public OfferMove(List<Agent> hearers, Agent speaker, Offer offer,
int id) {
			super(hearers, speaker, Act.Offer, offer, id);
		}

		@Override
		public String strBody() {
			return "Offer(" + offer + ")";
```

```
		}

		@Override
		public int hashCode() {
			return super.offer == null ? 0 : super.offer.hashCode();
		}

		@Override
		public boolean equals(Object obj) {
			if (this == obj) return true;
			if (!(obj instanceof OfferMove)) return false;
			OfferMove other = (OfferMove)obj;
			if (other == null) return false;
			if (offer != null && offer.equals(other.offer) || offer ==
other.offer)
				return true;
			return false;
		}
	}

	/**
	 * Represents an acceptance.
	 */
	public static class Accept extends Challengable {
		List<Challengable> acceptedMoves;

		public Accept(Agent hearer, Agent speaker, Challengable move, int
id) {
			super(Collections.singletonList(hearer), speaker, Act.Accept,
move.getOffer(), id);
			acceptedMoves = Collections.singletonList(move);
			isAgainst = move.isAgainstOffer();
		}

		Accept(Set<Agent> hearers, Agent speaker, Collection<Argue> moves,
int id) {
			super(new ArrayList<Agent>(hearers), speaker, Act.Accept,
null, id);
			acceptedMoves = new ArrayList<Challengable>(moves);
		}

		public List<Challengable> getAcceptedMoves() { return
Collections.unmodifiableList(acceptedMoves); }

		@Override
		protected String strBody() {
			return "Accept(" + Move.strList(acceptedMoves) + ")";
		}
	}

	/**
	 * Represents an offer refusal.
	 */
	public static class Refuse extends Challengable {
		public Refuse(Agent hearer, Agent speaker, Offer offer, int id) {
			super(Collections.singletonList(hearer), speaker, Act.Refuse,
offer, id);
			isAgainst = true;
		}

		@Override
		public String strBody() {
			return "Refuse(" + offer + ")";
		}
	}
```

```
	/**
	 * Represents an argument move.
	 */
	public static class Argue extends Challengable {
		private Argument argument;
		private Challengable challenged;

		public Argue(List<Agent> hearers, Agent speaker, Challengable m,
Argument arg, int id) {
			super(hearers, speaker, Act.Argue, m.getOffer(), id);
			this.argument = arg;
			this.challenged = m;
			this.isAgainst = argument.isCon();
		}

		public Challengable getChallenged() { return challenged; }

		/**
		 * Returns the argument of this move.
		 */
		public Argument getArgument() { return argument; }

		@Override
		public String strBody() {
			return "Argue for move " + challenged.id + ": " + argument;
		}
	}

	public static class Withdraw extends Move {
		public Withdraw(List<Agent> hearers, Agent speaker, int id) {
			super(hearers, speaker, Act.Withdraw, id);
		}

		@Override
		public String strBody() {
			return "Withdraw";
		}
	}

	public static class Challenge extends Move {
		private Set<Challengable> challenged;

		public Challenge(Agent speaker, Set<Challengable> challenged, int
id) {
			super(new ArrayList<Agent>(Move.getSpeakers(challenged)),
speaker, Act.Challenge, id);
			this.challenged = new HashSet<Challengable>(challenged);
		}

		public Set<Challengable> getChallenged() { return
Collections.unmodifiableSet(challenged); }

		@Override
		public String strBody() {
			return "Challenge(" + Move.strList(challenged) + ")";
		}
	}

	public static class SayNothing extends Move {
		public SayNothing(Agent speaker, int id) {
			super(Collections.<Agent>emptyList(), speaker,
Act.SayNothing, id);
		}
```

```
		@Override
		public String strBody() {
			return "Say Nothing";
		}
	}
}
```

protoimpl.java

```
package com.gorbas.negotiation.amgoud.impl;

import com.gorbas.model.*;
import com.gorbas.negotiation.amgoud.Move.Challengable;
import com.gorbas.negotiation.amgoud.*;
import java.util.*;

/**
 * Negotiation protocol implementation adjusted to work with our UTASTAR-
generated model.
 * @author gorbas
 */
public class ProtoImpl extends Protocol {

	/**
	 * Initializes the Offers list with the given alternatives.
	 * @param alternatives
	 */
	public void setAlternatives(Collection<Alternative> alternatives) {
		offers = new ArrayList<DmOffer>(alternatives.size());
		for (Alternative alt : alternatives)
			offers.add(new DmOffer(alt));
	}

	/**
	 * Decision Maker as Agent.
	 */
	static class DmAgent extends Agent {
		DecisionMaker dm;
		Contest contest;

		public DmAgent(ProtoImpl p, DecisionMaker dm) {
			super(p);
			this.dm = dm;
			contest = p.contest;
		}

		@Override
		public boolean equals(Object obj) {
			if (obj instanceof DmAgent)
				return dm.equals(((DmAgent)obj).dm);
			return false;
		}

		@Override
		public int hashCode() {
			return dm.hashCode();
		}

		@Override
		public String toString() {
			return dm.getName() + " (" + dm.getId() + ")";
		}

		public static List<DmAgent> fill(ProtoImpl p, List<DecisionMaker>
dml) {
			List<DmAgent> l = new ArrayList<DmAgent>(dml.size());
			for (DecisionMaker dm : dml)
				l.add(new DmAgent(p, dm));
			return l;
		}

		@Override
```

```
		protected List<DmArgument> getArgumentsForOffer(Agent a, Offer o,
boolean against) {
			List<DmArgument> args = new ArrayList<DmArgument>();
			List<Criterion> goals = contest.getCriterionList();
			for (Argument arg : acquiredBeliefs) {
				if (arg.isCon() == against &&
arg.getSubject().equals(a))
					args.add((DmArgument)arg);
			}

			Alternative alt = ((DmOffer)o).getAlternative();
			ProtoImpl p = (ProtoImpl)negProtocol;

			for (Criterion c : goals) {
				DmArgument arg = p.new DmArgument(contest, (DmAgent)a,
alt, c);
				if (arg.against == against && arg.getStrength() > 0)
					args.add(arg);
			}
			return args;
		}

		/**
		 * Finds an acceptable argument for/against all of the accepted
moves.
		 * @param challenger The agent that needs to be persuaded.
		 * @param acceptedMoves The accepted moves.
		 * @param against True if arguments against the moves are required.
		 * @return
		 */
		private Argument findAcceptableArgument(Agent challenger,
Collection<Challengable> acceptedMoves, boolean against) {
			Argument a = null;
			for (Challengable am : acceptedMoves) {
				Argument ama = findAcceptableArgument(challenger, am,
against);
				if (ama == null) continue;
				if (a == null || ama.isPreferredTo(a))
					a = ama;
			}
			return a;
		}

		@Override
		protected Argument findAcceptableArgument(Agent challenger,
Challengable m, boolean against) {
			// If we've been challenged for something we accepted
earlier, then present an argument in favor of that something.
			if (m.getAct() == Act.Accept)
				return findAcceptableArgument(challenger,
((Move.Accept)m).getAcceptedMoves(), against);

			DmOffer offer = (DmOffer)m.getOffer();
			if (m.getAct() == Act.Refuse)
				against = !against;

			List<DmArgument> args = getArgumentsForOffer(challenger,
offer, against);
			List<DmArgument> oppArgs = getArgumentsForOffer(challenger,
offer, !against);
			args.removeAll(usedArguments);

			if (args.isEmpty()) // We are out of arguments.
				return null;
```

```
                        List<DmArgument> acceptable = new
ArrayList<DmArgument>(args.size());
                        for (DmArgument arg : args) {
                               // We want acceptable arguments that do not already
exist in the set of used arguments.
                               if (arg.isAcceptable(oppArgs))
                                      acceptable.add(arg);
                        }
                        // Return the most preferred among the acceptable arguments.
                        return acceptable.isEmpty() ? null :
Argument.findMostPreferred(args);
                }

                @Override
                protected boolean isAcceptable(Argument a) {
                        List<DmArgument> oppositeArgs =
getArgumentsForOffer(a.getSubject(), new DmOffer(((DmArgument)a).alternative),
a.isPro());
                        return ((DmArgument)a).isAcceptable(oppositeArgs);
                }
        }

        /**
         * Alternative as Offer.
         */
        public static class DmOffer implements Offer {
                private Alternative alternative;

                public DmOffer(Alternative a) {
                        this.alternative = a;
                }

                public Alternative getAlternative() { return alternative; }

                @Override
                public boolean equals(Object obj) {
                        return (obj instanceof DmOffer &&
alternative.equals(((DmOffer)obj).alternative));
                }

                @Override
                public int hashCode() {
                        return alternative.hashCode();
                }

                @Override
                public String toString() {
                        return alternative.getDescription() + " (" +
alternative.getId() + ")";
                }
        }

        /**
         * Negotiation arguments based on UTASTAR output.
         */
        class DmArgument extends Argument {
                Alternative alternative; // The alternative which this argument
refers to.
                Criterion goal;
                DecisionMaker decisionMaker;
                DmAgent dmAgent;
                double avgCriterionUtility; // Criterion utility average over all
alternatives.
                double avgAlternativeUsage; // Average alternative usage over all
decision makers.
```

```
		double critUtility; // Criterion utility for the alternative of the
argument.
		double altUsage; // Alternative usage for the criterion (goal) of
the argument.
		Double weight;
		boolean against;

		public DmArgument(Contest contest, DmAgent dm, Alternative alt,
Criterion goal) {
			this.alternative = alt;
			this.goal = goal;
			this.dmAgent = dm;
			this.decisionMaker = dm.dm;
			this.avgCriterionUtility =
ProtoImpl.this.getAvgCriterionUtility(goal);
			this.avgAlternativeUsage =
ProtoImpl.this.getAvgAlternativeUsage(alt);
			DmAlternativeCriterionUtilityValue cuv =
DmAlternativeCriterionUtilityValue.retrieveByAlternativeDecisionMakerCriterion(al
t, decisionMaker, goal);
			this.critUtility = cuv != null ? cuv.getUtilityValue() : 0.0;
			DecisionMakerAlternativeUsage dmu =
DecisionMakerAlternativeUsage.retrieve(decisionMaker, alt);
			this.altUsage = dmu != null ? dmu.getUsageValue() : 0.0;
			this.against = critUtility < avgCriterionUtility; //altUsage
< avgAlternativeUsage;
		}

		/*
		 * Methods needed for hashset inclusion tests.
		 */

		@Override
		public boolean equals(Object obj) {
			if (obj instanceof DmArgument) {
				DmArgument o = (DmArgument)obj;
				return against == o.against
					&& alternative.equals(o.alternative)
					&& goal.equals(o.goal)
					&& decisionMaker.equals(o.decisionMaker);
			}
			return false;
		}

		@Override
		public int hashCode() {
			return (against ? 1 : 0) ^ alternative.hashCode() ^
goal.hashCode() ^ decisionMaker.hashCode();
		}

		@Override
		public String toString() {
			return "[weight=" + getWeight() + ", CU=" + critUtility + ",
avgCU=" + avgCriterionUtility + "] "
				+ "According to " + decisionMaker.getName() + " (" +
decisionMaker.getId() + ")"
				+ ", alternative " + alternative.getDescription() + "
(" + alternative.getId() + ")"
				+ " is " + (isPro() ? "in favor of" : "against") + "
goal " + goal.getName() + " (" + goal.getId() + ")";
		}

		/**
		 * Decides whether this argument is acceptable when compared to a
set of arguments of the opposite kind.
```

```
		 * @param otherArguments
		 * @return True if this argument overweights all other arguments.
		 */
		public boolean isAcceptable(Collection<? extends Argument>
otherArguments) {
			double w = this.getWeight();
			for (Argument oa : otherArguments)
				if (w < oa.getWeight())
					return false;
			return true;
		}

		@Override
		public Agent getSubject() {
			return dmAgent;
		}

		@Override
		public boolean isPro() {
			return !against;
		}

		@Override
		public double getLevel() {
			return against ? 0.0 : 1.0; // We are certain about all
arguments.
		}

		@Override
		public double getWeight() {
			// If the argument is against a goal just reverse the weight.
			if (null == weight) {
				weight = against ? m(critUtility) : critUtility;
			}
			return weight;
		}
	}

	private Contest contest;
	private List<DmOffer> offers;
	private Map<Alternative, Double> avgAltUsages = null;
	private Map<Criterion, Double> avgCritUtils = null;
	private Map<DecisionMaker, Map<Criterion, Double>> avgCritUtilsOverAlt =
null;
	private Map<DecisionMaker, Map<Alternative, Double>> avgCritUtilsOverCrit
= null;

	/**
	 * Returns the average alternative usage over all decision makers.
	 * @param alt
	 * @return
	 */
	private double getAvgAlternativeUsage(Alternative alt) {
		if (null == avgAltUsages) {
			avgAltUsages = new HashMap<Alternative, Double>();
			List<DecisionMakerAlternativeUsage> ul =
DecisionMakerAlternativeUsage.retrieveAverage(contest);
			for (DecisionMakerAlternativeUsage u : ul)
				avgAltUsages.put(u.getAlternative(),
u.getUsageValue());
		}
		Double usage = avgAltUsages.get(alt);
		return null == usage ? 0.0 : usage;
	}
```

```
	private double getAvgCriterionUtility(Criterion goal) {
		if (null == avgCritUtils)
			avgCritUtils = new HashMap<Criterion, Double>();
		Double u = avgCritUtils.get(goal);
		if (null == u) {
			u =
DmAlternativeCriterionUtilityValue.AverageUtilityValue(goal);
			avgCritUtils.put(goal, u);
		}
		return u;
	}

	private double getAvgCriterionUtility(DecisionMaker dm, Alternative alt) {
		if (null == avgCritUtilsOverCrit)
			avgCritUtilsOverCrit = new HashMap<DecisionMaker,
Map<Alternative, Double>>();

		Map<Alternative, Double> dmm = avgCritUtilsOverCrit.get(dm);
		if (null == dmm) {
			dmm = new HashMap<Alternative, Double>();
			avgCritUtilsOverCrit.put(dm, dmm);
		}

		Double u = dmm.get(alt);
		if (u == null) {
			u =
DmAlternativeCriterionUtilityValue.AverageUtilityValueForDmAlt(dm, alt);
			dmm.put(alt, u);
		}

		return u;
	}

	private double getAvgCriterionUtility(DecisionMaker dm, Criterion goal) {
		if (null == avgCritUtilsOverAlt)
			avgCritUtilsOverAlt = new HashMap<DecisionMaker,
Map<Criterion, Double>>();

		Map<Criterion, Double> dmm = avgCritUtilsOverAlt.get(dm);
		if (null == dmm) {
			dmm = new HashMap<Criterion, Double>();
			avgCritUtilsOverAlt.put(dm, dmm);
		}

		Double u = dmm.get(goal);
		if (u == null) {
			u =
DmAlternativeCriterionUtilityValue.AverageUtilityValueForDecisionMakerCriterion(d
m, goal);
			dmm.put(goal, u);
		}

		return u;
	}

	private double getUtility(DecisionMaker dm, Alternative a) {
		try {
			DecisionMakerAlternativeUsage dmau =
DecisionMakerAlternativeUsage.retrieve(dm, a);
			if (null != dmau)
				return dmau.getUsageValue();
		} catch (Exception ex) {}
		return 0;
	}
```

```
	public ProtoImpl(Contest c) {
		this.contest = c;
		this.agents = DmAgent.fill(this, c.getDecisionMakerList());
	}

	@Override
	public List<DmOffer> getOffers() {
		if (null == offers) {
			List<Alternative> alts = Alternative.retrieve();
			offers = new ArrayList<ProtoImpl.DmOffer>(alts.size());
			for(Alternative a : alts)
				offers.add(new DmOffer(a));
		}
		return offers;
	}

	@Override
	protected void removeOffer(Offer x) {
		getOffers().remove(x);
	}

	/**
	 * Implements offer preference decision based on the total utility value
of the offers for the agent.
	 * @param a
	 * @param b
	 * @param agent
	 * @param optimistic Ignored.
	 * @return True if alternative a has higher or equal utility value over b.
	 */
	@Override
	public boolean isAPreferredOverB(Offer a, Offer b, Agent agent, boolean
optimistic) {
		DecisionMaker dm = ((DmAgent)agent).dm;
		return getUtility(dm, ((DmOffer)a).getAlternative()) >=
getUtility(dm, ((DmOffer)b).getAlternative());
	}
}
```