Negotiation User Guide

T. Baarslag, W. Pasman, K. Hindriks, D. Tykhonov, W. Visser, M. Hendrikx

January 7, 2014

Abstract

GENIUS [**?**] is a negotiation environment that implements an open architecture for heterogeneous negotiating agents. GENIUS can be used to implement, or simulate, real life negotiations. This document describes how you can install the environment, work with the provided scenarios and negotiation agents, and write, compile, and run an agent yourself.

Contents

1 Theory Crash Course

This section provides a crash course on some essential theory needed to understand the negotiation system. Furthermore, it provides an overview of the features of a negotiation implemented in GENIUS.

1.1 Negotiation Protocol

The negotiation protocol determines the overall order of actions during a negotiation. Agents are obliged to stick to this protocol, as deviations from the protocol are caught and penalized. This section discusses the details of the bilateral alternating offers protocol used in GENIUS.

In the bilateral alternating offers protocol two parties – agent A and agent B – take turns. Agent A starts the negotiation. Each turn an agent presents one of the three possible actions:

ACCEPT	This action indicates that agent accepts the opponent's last bid.
OFFER	This action represents the bid made by an agent.
ENDNEGOTIATION	This action indicates that the agent terminates the negotiation.

When it is an agent's turn, it is informed about the opponent's action. Based on the opponent's action the agent comes up with a action, which it presents to the opponent. Sequentially, the opponent presents a counter action. This process goes on until the negotiation finishes in one of the following ways:

- An agent accepts the opponent's offer using the action ACCEPT. The utility of the opponent's last bid is determined for both agents according to their preference profiles. The opponent is informed of acceptance via the *ReceiveMessage* method.
- The action returned by an agent is ENDNEGOTIATION. In this case the score of both agents is set to their reservation value.
- Finally, if an agent does not follow the protocol – for instance by sending an action that is not one of the above or by crashing – the agent's utility is set to its reservation value, whereas the opponent is awarded the utility of the last offer.

1.2 Reservation Value

A reservation value is a real-valued constant that sets a threshold below which a rational agent should not accept any offers. Intuitively, a reservation value is the utility associated with the Best Alternative to a Negotiated Agreement (BATNA).

A reservation value is the utility that an agent will obtain if no agreement is realized in a negotiation session. This can happen either if an agent leaves the negotiation, or by not reaching an agreement before the deadline. In other words: either the negotiating parties agree on an outcome ω, and both agents receive the associated utility of ω, or no agreement is reached, in which case both agents receive their reservation value instead. Reservation values typically differ for each negotiation agent. In case no reservation value is set in a profile, it is assumed to be 0.

1.3 Time Pressure

A negotiation lasts a predefined time in seconds, or alternatively rounds. In GENIUS the time line is *normalized*, i.e.: time $t \in [0, 1]$, where $t = 0$ represents the start of the negotiation and $t = 1$ represents the deadline. Notice that manipulation of the remaining time can be a factor influencing the outcome.

There is an important difference between a time-based and rounds-based protocol. In a time-based protocol the computational cost of an agent should be taken into account as it directly influences the amount of bids which can be made. In contrast, for a rounds-based negotiation the time can be thought of as paused within a round; therefore computational cost does not play a role.

Apart from a deadline, a scenario may also feature *discount factors*. Discount factors decrease the utility of the bids under negotiation as time passes. While time is shared between both agents, the discount generally differs per agent. The implementation of discount factors is as follows: let d in $[0, 1]$ be the discount factor that is specified in the preference profile of an agent; let t in $[0, 1]$ be the current

normalized time, as defined by the timeline; we compute the discounted utility U_D^t of an outcome ω from the undiscounted utility function U as follows:

$$U_D^t(\omega) = U(\omega) \cdot d^t \tag{1}$$

If $d = 1$, the utility is not affected by time, and such a scenario is considered to be undiscounted, while if d is very small there is high pressure on the agents to reach an agreement. Note that discount factors are part of the preference profiles and therefore different agents may have a different discount factor.

If a discount factor is present, reservation values will be discounted in exactly the same way as the utility of any other outcome. It is worth noting that, by having a discounted reservation value, it may be rational for an agent to end the negotiation early and thereby default to the reservation value.

1.4 Negotiation Objects

Agents participating in a negotiation interact in a scenario. A scenario specifies the possible bids and their preference for both agents. A scenario consists of a domain (also called the outcome space) and two utility spaces (also called preference profiles). Figure **??** provides an overview of the relation between the domain and the utility space of an agent.

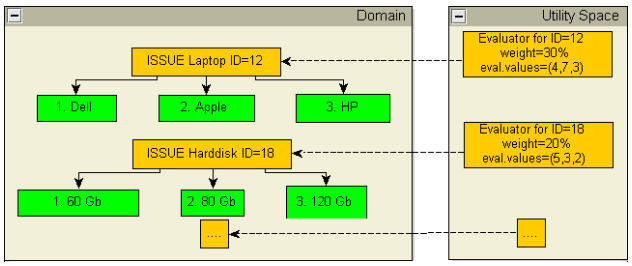

Figure 1: Overview of the data structures and relations.

The *Domain* describes which issues are the subject of the negotiation and which values an issue can attain. To give a concrete example of a domain: in the laptop domain the issues are "laptop", "harddisk" and "monitor". In the laptop domain the issues can only attain discrete values, e.g. the "harddisk" issue can only have the values "60 Gb", "80 Gb" and "120 Gb". These issues are all instance of *IssueDiscrete*.

Combining these concepts, an agent can formulate a *Bid*: a mapping from each issue to a value. A valid bid in the laptop domain is for example a Dell laptop with 80 Gb and a 17' inch monitor.

The *Utility Space* specifies the preferences of the bids for an agent. Using a utility space the utility of a bid can be calculated using the evaluator of each issue. The evaluator of an issue maps the evaluation of a value – which is specified in the preference profile – to a utility for that issue. The evaluator also specifies the importance of the issue relative to the other issues in the form of a weight. The weights of all issues sum up to 1.0 to simplify calculating the utility of a bid. To illustrate, the "harddisk" issue is of the type *IssueDiscrete*, and therefore its evaluator is of the type *EvaluatorDiscrete*.

In general, given the set of all bids, there are a small subset of bids which are more preferred as outcomes by both agents. Identifying these special bids may lead to a better agreement for both parties. We discuss the optimality of a bid in the next section.

1.5 Optimality of a Bid

Before discussing the optimality of a bid, we first need to formalize the concept of a bid. A bid is a set of chosen values $v_1, \ldots, v_n$ for each of the N issues. Each of these values has been assigned an evaluation value $\text{eval}(v_i)$ in the utility space. The utility is the weighted sum of the normalized evaluation values.

$$U(v_1, \ldots, v_n) = \sum_{i=1}^{N} w_i \frac{\text{eval}(v_i)}{\max(\text{eval}(v_i))} \tag{2}$$

For a single agent, the optimal bid is of maximum utility for the agent. Often this bid has a low utility for the opponent, and therefore the chance of agreement is low. A more general notion of optimality of a negotiation involves the utility of both agents.

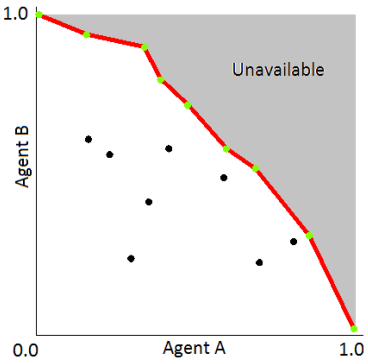

Figure 2: A point indicates the utility for both agents of a bid. The red line is the Pareto optimal frontier.

There are multiple ways to define a more global "optimum". One approach to optimality is that a bid is not optimal for both parties if there is another bid that has the higher utility for one party, and at least equal utility for the other party. Thus, only bids in Figure **??** for which there is no other bid at the top right is optimal. This type of optimality is called Pareto optimality and forms an important concept in automated negotiation. The collection of Pareto optimal bids is called the Pareto optimal frontier.

A major challenge in a negotiation using the bilateral alternating offers protocol is that agents hide their preferences. This entails that an agent does not know which bid the opponent prefers given a set of bids. This problem can be partly resolved by building a model of the opponent's preferences by analyzing the negotiation trace. Each turn the agent can now offer the best bid for the opponent given a set of similar preferred bids. By default, there are already a few opponent models implemented in GENIUS.

2 Running the Environment

The negotiation environment has been tested extensively on Microsoft Windows. It should run on any machine running Java 6 or higher, including Solaris and Linux distributions. Under Ubuntu the jar file should be launched from the terminal to avoid problems with finding the repository files. Please report any bugs found to `ai@mmi.tudelft.nl`.

To install the environment, the file `negotiator.zip` can be downloaded. Unzip the file at a convenient location on your machine. This will result in a package containing the following files:

- `userguide.pdf`, this document;
- `negosimulator.jar`, the negotiation simulator;
- a `templates` folder, containing various scenarios.

When you run the negosimulator (by double-clicking the application or using open with and then selecting Java), progress messages and error messages are printed mainly to the standard output. On Mac OSX you can view these messages by opening the console window (double-click on Systemdisk/Applications/Utilities/Console.app). On Windows this is not directly possible. Console output can be read only if you start the application from the console window by hand, as follows. Go to the directory with the negosimulator and enter `java -jar negosimulator.jar`. This will start the simulator, and all messages will appear in the console window. You may see some errors and warnings that are non-critical.

Note that some agents and scenarios require more memory than allocated by default to Java. This problem can be resolved by using the Xmx and Xms parameters when launching the executable jar, for example `java -Xmx1536M -Xms1536M -jar negosimulator.jar`.

3 Scenario Creation

A negotiation can be modeled in GENIUS by creating a scenario. A scenario consists of a domain specifying the possible bids and a set of preference profiles corresponding to the preferences of the bids in the domain. This section discusses how to create a domain and a preference profile.

3.1 Basic GUI Components

Start GENIUS by following the instructions in the previous section. After starting the simulator a screen similar to Figure **??** is shown. This screen is divided in three portions:

- The **Menubar** allows us to start a new negotiation.
- The **Components Window** shows all available scenarios, agents, and BOA components.
- The **Status Window** shows the negotiation status or selected domain/preference profile.

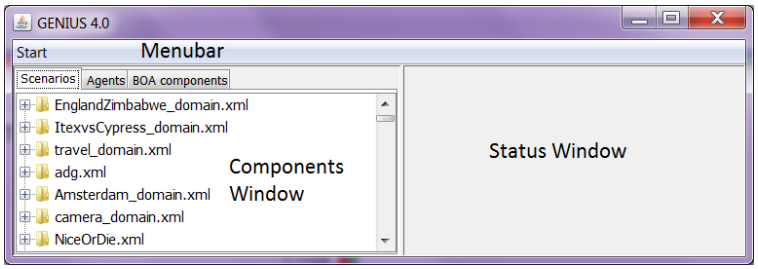

Figure 3: The negosimulator right after start-up.

3.2 Creating a Domain

By right clicking on the list of available scenarios in the Components Window a popup menu with the option to create a new domain is shown. After clicking this option it is requested how the domain should be called. Next the domain is automatically created and a a window similar to Figure **??** is shown. Initially, a domain contains zero issues. We can simply add an issue by pressing the "Add issue" button. This results in the opening of a dialog similar to Figure **??**.

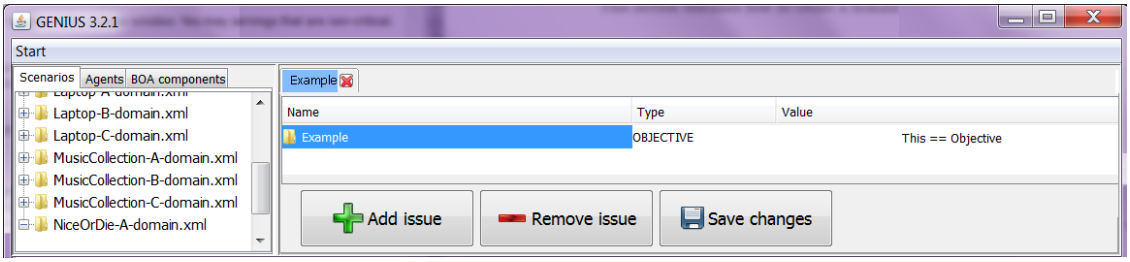

Figure 4: The negosimulator after loading the laptop domain.

The current version of GENIUS supports the creation of discrete and integer issues. Starting with a discrete issue, the values of the issue should be specified. In Figure **??** we show the values of the issue "Harddisk". Note the empty evaluation values window, later on when creating a preference profile we will use this tab to specify the preference of each value.

Instead of a discrete issue, we can also add an integer issue as shown in Figure **??**. For an integer issue we first need to specify the lowest possible value and the highest value, for example the price range for a second hand car may be $[700, 900]$. Next, when creating a preference profile we need to specify the

utility of the lowest possible value (700) and the highest value (900). During the negotiation we can offer any value for the issue within the specified range.

The next step is to press "Ok" to add the issue. Generally, a domain consists of multiple issues. We can simply add the other issues by repeating the process above. If you are satisfied with the domain, you can save it by pressing "Save changes".

Finally, note that the issues of a domain can only be edited if the scenario does not (yet) specify preference profiles. This is to avoid inconsistencies between the preference profiles and the domains.

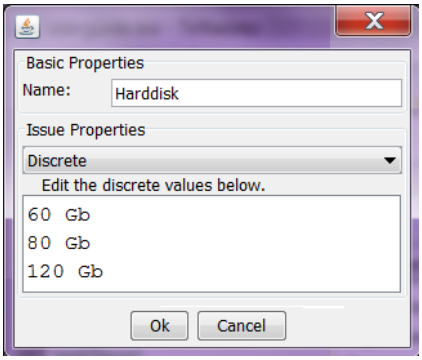

Figure 5: Creating a discrete issue.

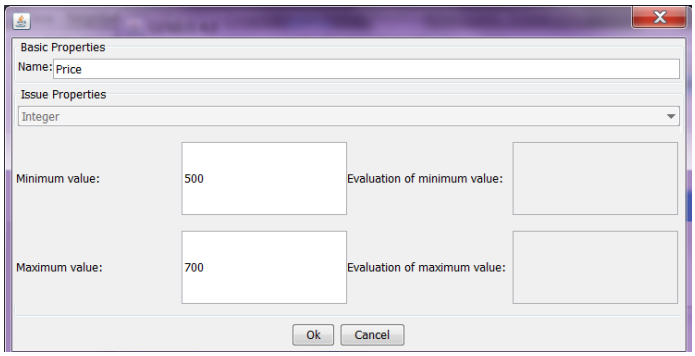

Figure 6: Creating an integer issue.

3.3 Creating a Preference Profile

Now that we created a domain, the next step is to add a set of preference profiles. By right clicking on the domain a popup menu is opened which has an option to create a new preference profile. Selecting this option results in the opening of a new window which looks similar to Figure **??**.

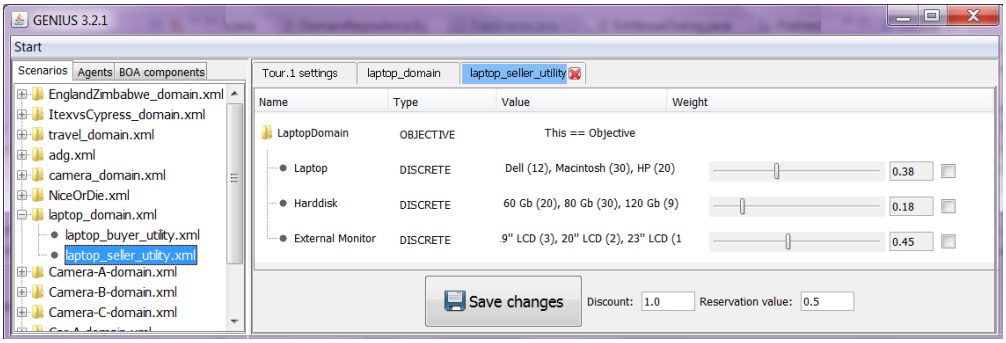

Figure 7: The negosimulator after creating a new utility space.

Now you are ready to start customizing the preference profile. There are three steps: setting the importance of the issues, determining the preference of the values of the issues, and configuring the reservation value and discount. To start with the first step, you can adjust the relative weights of the issues by using the sliders next to that issue. Note that when you move a slider, the weights of the other sliders are automatically updated such that the all weights still sum up to one. If you do not want that the weight of another issue automatically changes, you can lock its weight by selecting the checkbox behind it. Now that we set the weights of the issues, it is a good idea to save the utility space.

The next and final step is to set the evaluation of the issues. To specify the evaluation of an issue you can double click it to open a new window looking similar to Figure **??** or Figure **??** depending on the type of the issue.

For a discrete issue we need to specify the evaluation value of each discrete value. A specific value can be assigned any positive non-zero integer as evaluation value. During the negotiation the utility of

a value is determined by dividing the value by the highest value for that particular issue. To illustrate, if we give 60 Gb evaluation 5, 80 Gb evaluation 8, and 120 Gb evaluation 10; then the utilities of these values are respectively 0.5, 0.8, and 1.0.

Specifying the preference of a integer issue is even easier. In this case we simply need to specify the utility of the lowest possible value and the highest possible value. The utility of a value in this range is calculated during the negotiation by using linear interpolation of the utilities of both given utilities.

The final step is to set the reservation value and discount of a preference profile. If you are satisfied with the profile you can save it by pressing "Save changes". Finally, you can create additional preference profiles for the domain and run a negotiation.

4 Running Negotiations

This section discusses how to run a negotiation. There are three main modes to run a negotiation:

- **Negotiation session**. A negotiation session concerns a single negotiation in which two agents compete. This mode is mainly intended for new users.
- **Tournament**. A tournament is a collection of sessions. Two sets of agents compete against each other on a set of domains. The results of the sessions are stored in the "log" directory. These results can be more easily viewed by importing them into Excel and using pivot tables (cf. Section **??**).
- **Distributed tournament**. A distributed tournament is a tournament which is stored in a database and can therefore be divided among multiple computers to speed up calculation.

Before going into detail on how each of these modes work, we first discuss the two types of agents that can be used: automated agents and non-automated agents. Automated agents are agents that can compete against other agents in a negotiation without relying on input by a user. In general, these agents are able to make a large amount of bids in a limited amount of time.

In contrast, non-automated agents are agents that are fully controlled by the user. These types of agents ask the user each round which action they should make. GENIUS by default includes the UIAgent – which has a simple user interface – and the more extensive Extended UIAgent.

4.1 Running a Negotiation Session

To run a negotiation session select "Start" and then "Negotiation Session". This opens a window similar to Figure **??**. The following parameters need to be specified to run a negotiation:

- **Negotiation protocol**. The set of available protocols. Normally "Alternating Offers" is used.
- **Side A/Side B**. The configuration of the agents of both sides.
- **Preference profile**. The preference profile to be used by the agent of that side.
- **Agent name**. The agent participating in the negotiation.
- **Deadline (seconds)**. The length of the negotiation in seconds.

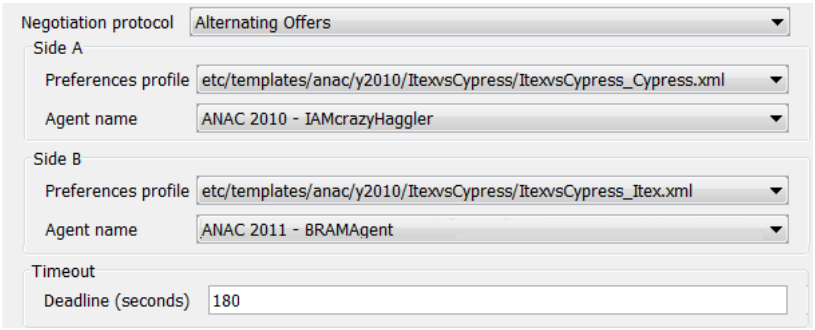

Figure 8: A negotiation session.

4.2 Running a Tournament

Besides running a single negotiation session, it is also possible to run a tournament. A tournament can be seen as a collection of sessions. In contrast to running a single session, the results of a tournament are stored in the "log" directory. These results can be easily analyzed by importing them into Excel (cf. Section **??**). A tournament can be created by first selecting "Start" and then "Tournament". The Tournament tab will appear similar to Figure **??**. This window shows a set of options which we need to specify. The value of an option can be specified by double clicking the option in the "Values" column.

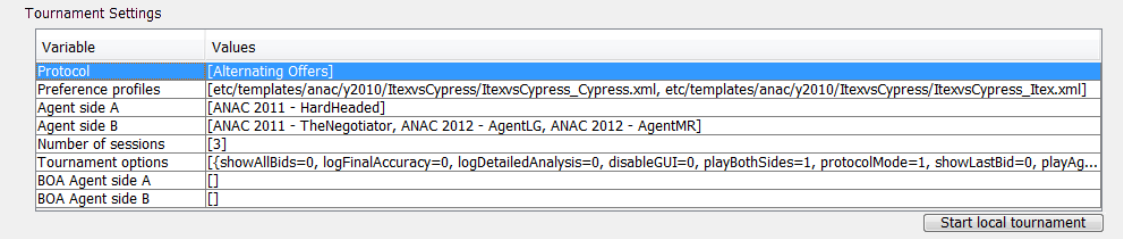

Figure 9: Tournament tab.

- **Protocol**. The set of available protocols.
- **Preference profiles**. The set of scenarios on which the agents should compete. Each selected scenario should feature at least two preference profiles.
- **Agent side A/B**. The set of agents in set A competes against all agents in set B.
- **Number of sessions**. The number of times each session should be repeated.
- **Tournament options**. Options which specify how to run the tournament (see below).
- **BOA Agent side A/B**. Type of agents that consist of multiple components (see Section **??**).

A large set of tournament options can be specified which influence the composition and running of the tournament. There are four categories of options:

- Protocol settings
  - **Protocol mode**. Specifies if the negotiation features rounds or time. In a time-based negotiation there is an amount of time to reach an agreement. Time passes while an agent deliberates an action. In contrast, in a rounds-based negotiation the deadline is specified in rounds. An agent can take more time to compute an action as time does not pass within a round.
  - **Deadline**. Depending on the protocol mode, this is the maximum amount of time in seconds or amount of rounds. Note that one single round corresponds to one turn of a single agent.
  - **Access partner preferences**. Allows agents to access the preference profile of the negotiation session, which contains the opponent's preference profile.
  - **Allow pausing timeline**. Allow agents to pause the negotiation by using the timeline.pause() and time.resume() methods.
- Session generation
  - **Play both sides**. When generating the sessions, whether each pair of agents should play both sides on a scenario or not.
  - **Play against self**. An agent may be included both in the set Agent side A and side B. If this option is enabled an agent is allowed to play against itself. If disabled, the sessions in which agents negotiate against themselves are removed.
- Logging
  - **Log detailed analysis**. Enabling this option activates a set of quality measures to capture the quality of the negotiation process. The quality measures are added to the default log. In addition, for the whole tournament an overview log is created. This log is prefixed with "TM-".

- Visualization
  - **Show all bids**. When enabled all bids in a scenario are visualized as red points in the negotiation status window. This option has some impact on performance.
  - **Show last bid**. When enabled the last bid is marked with a special symbol to make it clear which move an agent performed.
  - **Disable GUI**. When enabled most GUI elements are disabled. This speeds-up the negotiation up to a factor of 200 times. The progress of the tournament is printed to the console.

4.3 Advanced: Running a Distributed Tournament

A tournament quickly becomes practically too large to run. Running a distributed tournament resolves this problem as the tournament is stored in a database. Next, instances of Genius – perhaps running on the same computer – can connect to the database and process part of the tournament.

Before we can run a distributed tournament, we first need to setup a simple MySQL server which can be accessed by the computers. The installation of the database should include the "InnoDB" database engine. We will use this engine because it allows us to more easily remove old tournament data that we no longer need. Furthermore we recommend at least 50 Mb of free space. The required database structure can be created by using the SQL dump which can be found in the directory *doc/database*.

The next step is to specify a tournament to run. Towards this end, select "Start" and then "Distributed tournament". This opens a GUI similar to Figure **??**, except for the following four options:

- **Database address**. The address of the database, for example `sql.ewi.tudelft.nl:3306/DG`.
- **Database user**. The username of the account for the database.
- **Database password**. The password of the user account for the database.
- **Database sessionname**. The identifier of the tournament. The identifier is needed as multiple distributed tournaments can be run at the same time.

After specifying the tournament and database parameters we can start the distributed tournament by pressing "Start distributed tournament". Selecting this button splits the tournament into smaller jobs which are stored in the database. The tournament is automatically started similar to a normal tournament. Now other computers can easily connect by specifying the database parameters and selecting "Join distributed tournament". For these computers we only need to fill in the database parameters as the configuration is loaded from the database. Finally, after running the full tournament the results are sent to all computers and stored in the "log" directory. Figure **??** summarizes the process.

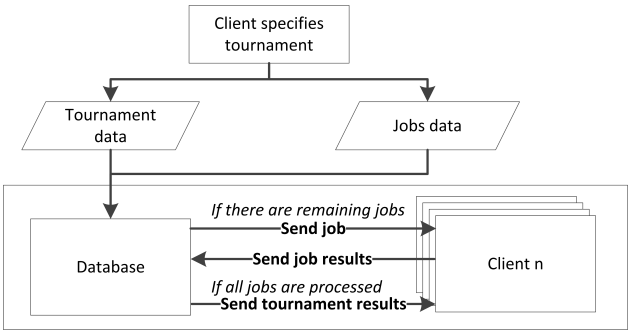

Figure 10: Distributed tournament process.

It should be noted that currently there is no option in Genius to delete old tournament data. Therefore we recommend to install *phpMyAdmin*. Using phpMyAdmin the old data of a tournament can be easily deleted by removing the tournament in the jobs table.

5 Quality Measures in Genius

A large set of quality measures have been incorporated in GENIUS 4.0. Most quality measures are automatically available, while for others an option must be selected in the tournament options menu.

There are now two types of logs used in GENIUS: the standard log and the tournament log. The standard log captures the outcome of each negotiation in a tournament by logging the results of the quality measures for both agents. The tournament log uses the standard log to calculate averages and standard deviations of functions of the quality measures in the standard log, for example the average final utility for all sessions which resulted in an agreement.

First, Section **??** discusses the measures incorporated in the standard log. Next, Section **??** details the tournament log. Finally, Section **??** discusses how Excel can be used to analyze logs.

5.1 Overview of Quality Measures in the Standard Log

GENIUS 4.0 incorporates two types of quality measures: standard measures and detailed measures. In addition there are some experimental measure types, such as competitiveness and opponent model accuracy, however these are not discussed here. In the following sections we discuss both measure types in detail.

5.1.1 Standard Measures

The standard measures are the measures which are enabled by default and cannot be disabled. Table **??** provides an overview of all default quality measures.

Attribute	Description
acceptance_strategy	The acceptance strategy of a BOA agent (see Section **??**).
agent	The side at which the agent played (A or B).
agentClass	The classpath of the agent.
agentName	The name of the agent.
bestAcceptableBid	Utility of the best bid offered to the agent. Note that the discount is not taken into account.
bestDiscountedAccepableBid	Utility of the best bid offered to the agent, taking the discount into account.
bids	Amount of offers exchanged during the negotiation.
currentTime	Time of storage of the result of the negotiation.
discountedUtility	The discounted utility earned by the agent in the negotiation.
domain	Domain at which the negotiation took place.
errors	Errors encountered during the negotiation. Not reaching an agreement before the deadline is also treated as an error.
finalUtility	The undiscounted utility earned by the agent in the negotiation.
lastAction	Last action made before the negotiation ended.
normalized_utility	The final utility divided by the maximum possible utility according to the preference profile. In correct domains the result should be equal to the final utility.
offering_strategy	The offering strategy of a BOA agent (see Section **??**).
opponent-agentClass	The classpath of the opponent.
opponent-agentName	The name of opponent's agent.
opponent_model	The opponent model of a BOA agent (see Section **??**).
opponent-utilSpace	The opponent's preference profile.
runNumber	How many times the negotiation has been repeated before.
startingAgent	Side which started the negotiation: A or B.
timeOfAgreement	Normalized time at which an agreement was established. 1.0 for no agreement.
utilSpace	The agent's preference profile.

Table 1: Standard quality measures in GENIUS in alphabetic order.

5.1.2 Detailed Measures

The detailed quality measures consist of trajectory analysis measures and measures for the fairness and optimality of the outcome. The detailed measures can be enabled by selecting "Log detailed analysis" in the tournament options menu. Enabling this option also results in the generation of the tournament log discussed in Section **??**.

Attribute	Description
concession_moves	The percentage of moves in which the agent, relative to the previous offer, offered a bid with decreased its own utility and increased its opponent's utility.
exploration_rate	The percentage of bids in the outcome space explored by the agent. Two bids with exactly the same utilities for both parties are treated as a single same bid.
fortunate_moves	The percentage of moves in which the agent, relative to the previous offer, offered a bid which increased both its own and its opponent's utility.
joint_exploration_bids	The percentage of unique bids of the outcome space explored by both agents together. Two bids with exactly the same utilities for both parties are treated as a single same bid.
kalai_distance	Distance from the undiscounted utilities of the outcome to the Kalai-Smorodinsky solution.
nash_distance	Distance from the undiscounted utilities of the outcome to the Nash solution.
nice_moves	The percentage of moves in which the agent, relative to the previous offer, offered a bid which increased its opponent's utility without significantly changing its own utility.
pareto_distance	Distance from the undiscounted utilities of the outcome to the nearest bid on the Pareto-optimal frontier.
perc_pareto_bids	Percentage of Pareto-optimal bids offered by an agent.
selfish_moves	The percentage of moves in which the agent, relative to the previous offer, offered a bid which increased its own utility and decreased its opponent's utility.
silent_moves	The percentage of moves in which the agent, relative to the previous offer, offered a bid which which was (nearly) equally valued by both agents.
social_welfare	A fairness measure being the sum of the utilities for both agents.
unfortunate_moves	The percentage of moves in which the agent, relative to the previous offer, offered a bid which decreased both its own and its opponent's utility.

Table 2: Detailed quality measures in GENIUS in alphabetic order.

5.2 Overview of Quality Measures in the Tournament Log

The tournament log is an analysis of the results on the quality measures for each agent, for example the average utility for *Agent K*. Similar to the detailed quality measured the tournament log can be enabled by selecting "Log detailed analysis" in the tournament options menu.

Three types of measures are included in the log:

- **Averages of quality measures**. The tournament log includes a large set of averages of the quality measures in the standard log. Examples include the average Nash distance, the average percentage of silent moves, and the average social welfare.
- **Standard deviations of quality measures**. The tournament log also includes the standard deviation of some measures. Note that this not the normal standard deviation of for example the utility, but the more complicated deviation between runs. To illustrate, if there were ten runs of

the tournament, then each run has an average utility and we can calculate the standard deviation of this utility between runs.

- **Average of functions of quality measures**. The tournament log also includes a large set of measures which are functions of measures included in the standard log. An example is the average utility for an agent only for the matches which resulted in agreement.

5.3 Analyzing Logs using Excel

The logs are in XML format, which entails that we can easily analyze them by using Excel. Note that the following discussion does not apply to the starter edition of Excel, as it does not support Pivot tables.

The XML data of the standard log can be converted to a normal table by importing the data into Excel using the default options. This results in a large table showing the result for both agents A and B for each session. Analyzing these results manually is complicated, therefore we recommend to use pivot tables. Pivot tables allow to summarize a large set of data using statistics and can be created by selecting "Insert" and then "Pivot Table". To illustrate, by dragging the *agentName* in "Row Labels" and the *discountedUtility* in "Values" (see Figure **??**), we can easily see which agent scored best in the tournament. If solely the amount of matches of each agent is displayed, you need to set the "Value Field Settings" of *discountedUtility* to average instead of count.

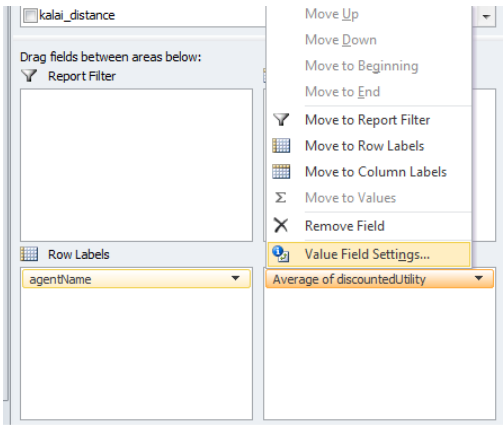

Figure 11: Configuration required to summarize the discounted utility of each agent.

6 Creating a Negotiation Agent

This section discusses how to create a basic negotiation agent. A standard negotiation agent implements an agent as a single block of logic: a mix of a bidding strategy, acceptance strategy, and possibly an opponent model. In contrast, we recommend to separately implement these components to create a BOA agent as discussed in Section **??**. The main advantage of a BOA agent is that existing components can be reused, allowing for easier agent development.

In this section we assume that you are familiar with programming in Java. In case you are not familiar with Java, please consult the following tutorial: `http://docs.oracle.com/javase/tutorial/index.html`. The Java API definitions can be found on `http://docs.oracle.com/javase/7/docs/api/index.html`.

The recommended way to create an agent is to create a new project in for example Eclipse or Netbeans. Next, add the *negotiationsimulator.jar* as an external library to the project such that classes in the project can use the classes of GENIUS. Finally, to create an agent create a new class and extend the *negotiator.Agent* class. Table **??** shows the most important fields and methods of this class. For more information, please refer to the javadoc of GENIUS. To implement your agent, you have to override the three methods: *ReceiveMessage*, *init*, and *chooseAction*. An agent may consist of multiple classes as long as one class extends the *negotiator.Agent* class.

`UtilitySpace utilitySpace` The preference profile of the scenario allocated to the agent.
`Timeline timeline` Use timeline for every time-related by using `getTime()`.
`double getUtility(Bid bid)` A convenience method to get the utility of a bid taking the discount factor into account.
`void init()` Informs the agent about beginning of a new negotiation session.
`void ReceiveMessage(Action opponentAction)` Informs the agent which action the opponent did.
`Action chooseAction()` This function should return the action your agent wants to make next.
`String getName()` Returns the name of the agent. Please override this to give a proper name to your agent.

Table 3: The most important methods and fields of the Agent class.

6.1 Receiving the Opponent's Action

The `ReceiveMessage(Action opponentAction)` informs you that the opponent just performed the action `opponentAction`. The `opponentAction` may be `null` if you are the first to place a bid, or an `Offer`, `Accept` or `EndNegotiation` action. The `chooseAction()` asks you to specify an `Action` to send to the opponent.

In the SimpleAgent code, the following code is available for `ReceiveMessage`. The SimpleAgent stores the opponent's action to use it when choosing an action.

```
public void ReceiveMessage(Action opponentAction) {
    actionOfPartner = opponentAction;
}
```

6.2 Choosing an Action

The code block below shows the code of the method `chooseAction` for SimpleAgent. For safety, all code was wrapped in a try-catch block, because if our code would accidentally contain a bug we still want to return a good action (failure to do so is a protocol error and results in a utility of 0.0).

The sample code works as follows. If we are the first to place a bid, we place a random bid with sufficient utility (see the .java file for the details on that). Else, we determine the probability to accept the bid, depending on the utility of the offered bid and the remaining time. Finally, we randomly accept or pose a new random bid.

While this strategy works, in general it will lead to suboptimal results as it does not take the opponent into account. More advanced agents try to model the opponent's strategy or preference profile.

```
public Action chooseAction() {
    Action action = null;
    try {
        if (actionOfPartner == null) {
            action = chooseRandomBidAction();
        }
        if (actionOfPartner instanceof Offer) {
            Bid partnerBid = ((Offer) actionOfPartner).getBid();
            double offeredUtilFromOpponent = getUtility(partnerBid);
            // get current time
            double time = timeline.getTime();
            action = chooseRandomBidAction();

            Bid myBid = ((Offer) action).getBid();
```

```
                double myOfferedUtil = getUtility(myBid);

                // accept under certain circumstances
                if (isAcceptable(offeredUtilFromOpponent,myOfferedUtil,time)) {
                    action = new Accept(getAgentID());
                }
            }
        } catch (Exception e) {
            e.printStackTrace();
            action = new Accept(getAgentID()); // best guess if things go wrong.
        }
        return action;
    }
```

The method *isAcceptable* implements the probabilistic acceptance function P_{accept}:

$$P_{\text{accept}} = \frac{u - 2ut + 2\left(t - 1 + \sqrt{(t-1)^2 + u(2t-1)}\right)}{2t - 1} \tag{3}$$

where u is the utility of the bid made by the opponent (as measured in our utility space), and t is the current time as a fraction of the total available time. Figure **??** shows how this function behaves depending on the utility and remaining time. Note that this function only decides if a bid is acceptable or not. More advanced acceptance strategies also use the `EndNegotiation` action.

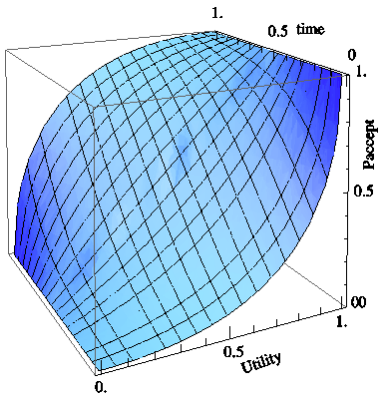

Figure 12: P_{accept} value as function of the utility and time (as a fraction of the total available time).

Automatic agents have to negotiate on their own, and are not allowed to communicate with a human user. Therefore, do not override the `isUIAgent()` function in automatic negotiation agents.

6.3 Overview of Classes

This section provides an overview of classes which might be useful when implementing an agent. For the documentation of the data structures that are presented, please refer to the Javadoc that can be found in your download of GENIUS.

- **BidDetails** is a structure to store a bid and its utility.
- **BidDetailsTime** is a structure to store a bid, its utility, and the time of offering.
- **BidHistory** is a structure to keep track of the bids presented by the agent and the opponent.
- **BidIterator** is a class used to enumerate all possible bids. Also refer to *SortedOutcomeSpace*.
- **BidSpace** is a class which can be used to determine the Pareto-optimal frontier and outcomes such as the Nash solution. This class can be used with the opponent's utility space as estimated by an opponent model.

- **Pair** is a simple pair of two objects.
- **Range** is a structure used to describe a continuous range.
- **SortedOutcomeSpace** is a structure which stores all possible bids and their utilities by using BidIterator. In addition, it implements efficient search algorithms that can be used to search the space of possible bids for bids near a given utility or within a given utility range.
- **UtilitySpace** is a representation of a preference profile. It is recommended to use this class when implementing a model of the opponent's preference profile.

6.4 Compiling an Agent

Compiling an agent can be done in two ways. First, in Eclipse you can simply find the compiled agent in the "bin" folder in your workspace. Alternatively, it is possible to manually compile the agent by placing `YourAgent.java` in the directory containing the `negotiator.jar` file, and using the command line: `javac -cp negosimulator.jar YourAgent.java`.

The next step is to load the agent in GENIUS. Towards this end, first we should move the agent to the directory of GENIUS. Note that the directory in which the agent should be put depends on your project. If for example you have a project with a package "agents" in which the agent is located, then the agent should be moved in a folder "agents" in the root of your GENIUS installation. An agent may consist of multiple class files. Now we can add the agent in one of the following two ways:

- **Loading the agent using the GUI**. An agent can be easily added by going to the "Agents" tab in the "Components Window" (see Figure **??**). Next, pressing right click opens a popup with the option to add a new agent. The final step is to select the main class of your agent.

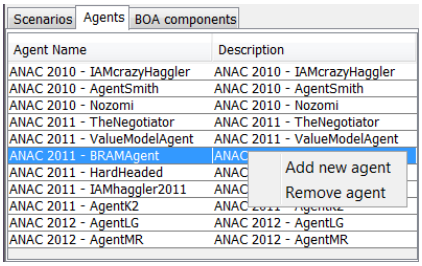

Figure 13: Adding an agent using the GUI.

- **Loading the agent using XML**. A compiled agent can also be loaded by directly adding the agent to the repository using the *agentrepository.xml* file. The code below visualizes a repository with a single agent. An agent element consists of several subelements; the first element is the *description* of the agent which is visualized in the GUI; the second element is the *classPath* specifying were the compiled agent class is located; the third element specifies the *agentName*; finally the optional element *params* specifies the parameters and their values available to the agent. In this case, a parameter "e" with value 2 and a parameter "time" with value 0.95 is specified. Variables can be accessed during the negotiation by using the *getStrategyParameters* method.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<repository fileName="agentrepository.xml">
  <items>
   <agentRepItems>
     <agentRepItem description="Other agents - SimpleAgent"
            classPath="agents.SimpleAgent"
            agentName="SimpleAgent" params="e=2;time=0.95"/>
     </agentRepItems>
  </items>
<filename>agentrepository.xml</filename>
</repository>
```

6.5 Creating a ANAC2013 Agent

The ANAC2013 introduces the concept that an agent can save and load information for each preference profile. This entails that an agent can learn from previous negotiations, against the same opponent or multiple opponents, to improve its competence when having a specific preference profile. It is only possible to retrieve information learned on the current preference profile as to avoid having perfect knowledge about the opponent's preferences.

A single serializable object can be saved per preference profile by using the *saveSessionData* method. If an object was already saved for the preference profile it is replaced. We recommend to store objects in the *endSession* method, which is called when a negotiation is finished. The saved object can be requested by using the *loadSessionData* method. A good place to do so is in in the *init* method. Note that this functionality is also available to BOA agents (cf. Section **??**).

7 Creating a BOA Agent

Instead of implementing your negotiating agent from scratch, we recommend you create a BOA agent using the *BOA framework*. The BOA negotiation agent architecture allows to reuse existing components from other BOA agents. Many of the sophisticated agent strategies that currently exist are comprised of a fixed set of modules. Generally, a distinction can be made between four different modules: one module that decides whether the opponent's bid is acceptable (*acceptance strategy*); one that decides which set of bids could be proposed next (*bidding strategy*); one that tries to guess the opponent's preferences (*opponent model*), and finally a component which specifies how the opponent model is used to select a bid for the opponent (*opponent model strategy*). The overall negotiation strategy is a result of the interaction between these components.

The advantages of separating the negotiation strategy into these four components (or equivalently, fitting an agent into the BOA framework) are threefold: first, it allows to *study the performance of individual components*; second, it allows to *systematically explore the space of possible negotiation strategies*; third, the reuse of existing components *simplifies the creation of new negotiation strategies*.

7.1 Components of the BOA Framework

A negotiation agent in the BOA framework, called a *BOA agent*, consists of four components:

Bidding strategy A bidding strategy is a mapping which maps a negotiation trace to a bid. The bidding strategy can interact with the opponent model by consulting with it.

Opponent model An opponent model is in the BOA framework a learning technique that constructs a model of the opponent's preference profile.

Opponent model strategy An opponent model strategy specifies how the opponent model is used to select a bid for the opponent and if the opponent model may be updated in a specific turn.

Acceptance strategy The acceptance strategy determines whether the opponent's bid is acceptable and may even decide to prematurely end the negotiation.

The components interact in the following way (the full process is visualized in Figure **??**). When receiving a bid, the BOA agent first updates the *bidding history*. Next, the *opponent model strategy* is consulted if the *opponent model* may be updated this turn. If so, the *opponent model* is updated.

Given the opponent's bid, the *bidding strategy* determines the counter offer by first generating a set of bids with a similar preference for the agent. The *bidding strategy* uses the *opponent model strategy* to select a bid from this set taking the opponent's utility into account.

Finally, the *acceptance strategy* decides whether the opponent's action should be accepted. If the opponent's bid is not accepted by the acceptance strategy, then the generated bid is offered instead.

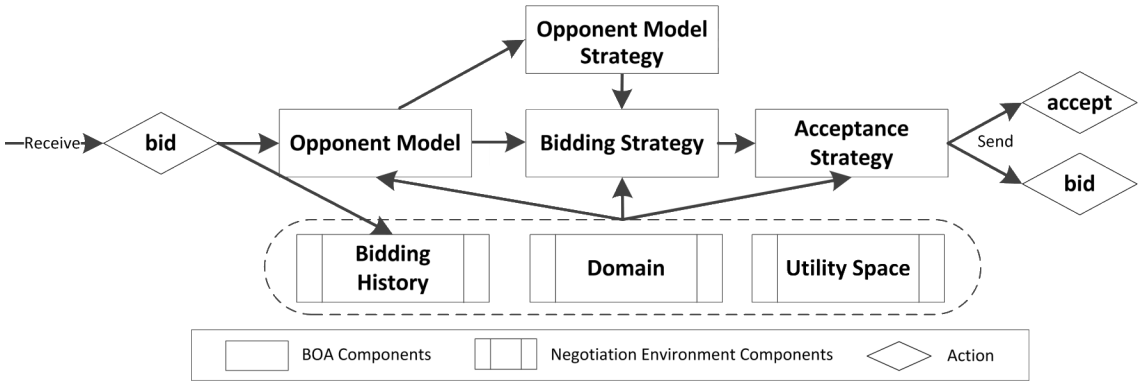

Figure 14: The BOA Framework Architecture.

7.2 Using Existing Components

In this section we create a *BOA agent* by selecting its components from a list of existing components. The BOA framework GUI (see Figure **??**) can be opened by double clicking the *Values* section next to the *BOA Agent side A* or *BOA Agent side B* when creating a (distributed) tournament.

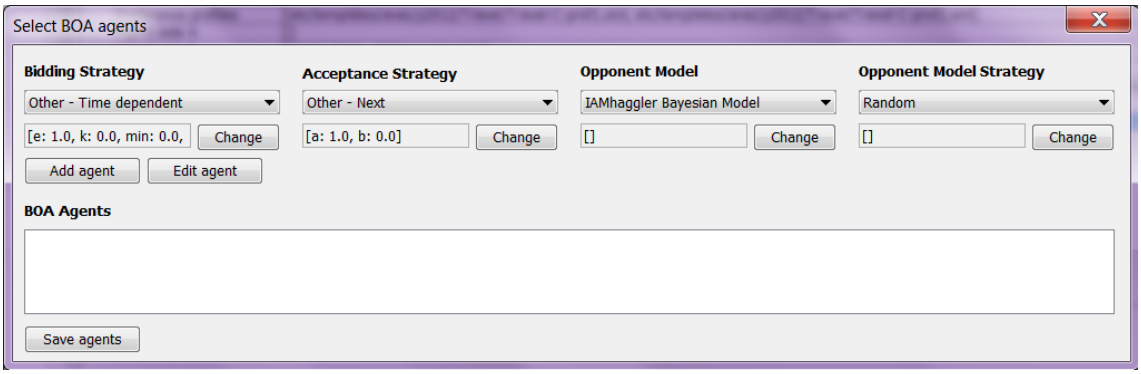

Figure 15: The BOA framework GUI.

Our goal in this section is to specify three BOA agents which are equal except for a single parameter a of their acceptance strategy. First, we select the bidding strategy *Other - Time Dependent* under the heading *Bidding Strategy*. Note that when we select this strategy, the default parameters of the component appear in the textbox below. Next, we select the other three components shown in Figure **??**.

The next step is to specify three variants of the acceptance strategy differing in the parameter a. To be more precise, we want a to be 1.0, 1.1, and 1.2. To achieve this, select "Change" to open a window similar to Figure **??**. Next, fill in the fields as shown in Figure **??**. Finally, we select "Add agent(s)" to create the three agents. Note that in this example we only varied a single parameter of a single component. If we vary more parameters possibly of different components, then all possible combinations are generated.

7.3 Creating New Components

This section discusses how create your own components. An example implementation of each component is included in the "BOA example" folder. The next section discusses how these components can be added to the list of available components in the BOA framework GUI.

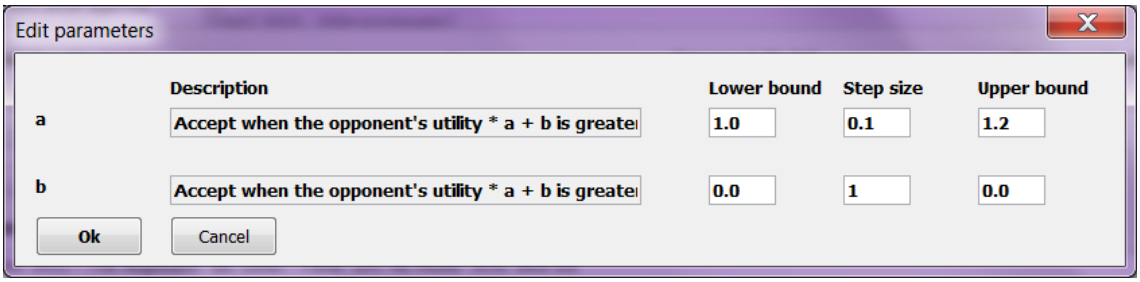

Figure 16: Adding a parameter.

7.3.1 Creating a Bidding Strategy

A bidding strategy can be easily created by extending the *OfferingStrategy* class. Table **??** depicts the methods which need to be overridden. The *init* method of the bidding strategy is automatically called by the BOA framework with four parameters: the negotiation session, the opponent model, the opponent model strategy, and the parameters of the component. The negotiation session object keeps track of the negotiation state, which includes all offers made by both agents, the timeline, the preference profile, and the domain. The parameters object specifies the parameters as specified in the GUI. In the previous section we specified the parameter b for the acceptance strategy $Other - Next$ to be 0.0. In this case the agent can retrieve the value of the parameter by calling *parameters.get("b")*.

An approach often taken by many bidding strategies is to first generate all possible bids. This can be efficiently done by using the *SortedOutcomeSpace* class. For an example on using this class see the *TimeDependent_Offering* class in the *BOA example* directory.

`void init(NegotiationSession negotiationSession, OpponentModel opponentModel, OMStrategy omStrategy, HashMap<String, Double> parameters)` Method directly called after creating the agent which should be used to initialize the component.
`BidDetails determineOpeningBid()` Method which determines the first bid to be offered to the component.
`BidDetails determineNextBid()` Method which determines the bids offered to the opponent after the first bid.

Table 4: The main methods of the bidding strategy component.

7.3.2 Creating an Acceptance Condition

This section discusses how to create an acceptance strategy class by extending the abstract class *AcceptanceStrategy*. Table **??** depicts the two methods which need to specified.

`void init(NegotiationSession negotiationSession, OfferingStrategy offeringStrategy, OpponentModel opponentModel, HashMap<String, Double> parameters)` Method directly called after creating the agent which should be used to initialize the component.
`Actions determineAcceptability()` Method which determines if the agent should accept the opponent's bid (*Actions.Accept*), reject it and send a counter offer (*Actions.Reject*), or leave the negotiation (*Actions.Break*).

Table 5: The main methods of the acceptance strategy component.

7.3.3 Creating an Opponent Model

This section discusses how to create an opponent model by extending the abstract class *OpponentModel*. Table **??** provides an overview of the main methods which need to specified. For performance reasons it is recommended to use the *UtilitySpace* class.

`void init(NegotiationSession negotiationSession, HashMap<String, Double> parameters)` Method directly called after creating the agent which should be used to initialize the component.
`double getBidEvaluation(Bid bid)` Returns the estimated utility of the given bid.
`double updateModel(Bid bid)` Updates the opponent model using the given bid.
`UtilitySpace getOpponentUtilitySpace()` Returns the opponent's preference profile. Use the *UtilitySpaceAdapter* class when not using the UtilitySpace class for the opponent's preference profile.

Table 6: The main methods of the opponent model component.

7.3.4 Creating an Opponent Model Strategy

This section discusses how to create an opponent model strategy by extending the abstract class *OM-Strategy*. Table **??** provides an overview of the main methods which need to specified.

`void init(NegotiationSession negotiationSession, OpponentModel model, HashMap<String, Double> parameters)` Method directly called after creating the agent which should be used to initialize the component.
`BidDetails getBid(List<BidDetails> bidsInRange);` This method returns a bid to be offered from a set of given similarly preferred bids by using the opponent model.
`boolean canUpdateOM();` Determines if the opponent model may be updated this turn.

Table 7: The main methods of the opponent model strategy component.

7.4 Adding a Component to the BOA Repository

In the previous section we discussed how to create each type of BOA component. To use the components, we still need to add them to the *BOA repository*. To do so, simply open the BOA components tab in the components window as shown in Figure **??** and select "Add new component".

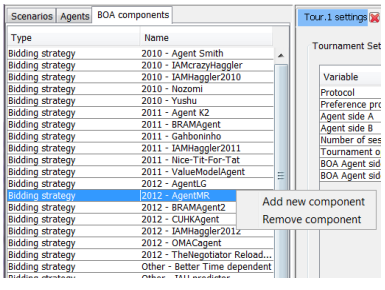

Figure 17: The BOA components window.

Selecting "Add new component" results in the opening of the window shown in Figure **??**. In this window the user should specify the name of the component, for example "ANAC2012 - AgentLG", specify the path to the class by selecting "Open"; and optionally add parameters. Finally, clicking "Add component" in this window adds the component to the repository.

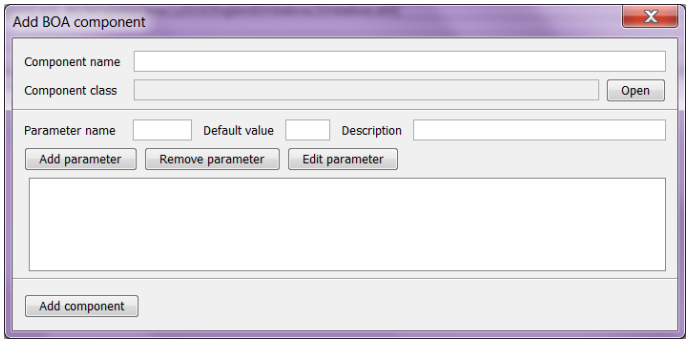

Figure 18: Loading a BOA agent.

7.5 Creating a ANAC2013 BOA Agent

In Section **??** we discussed how to create an agent for the ANAC2013. Using a similar procedure it is also possible to create BOA agents compatible with ANAC2013. An example to do so is included in this distribution of GENIUS.

As only a single object can be saved and loaded, the BOA framework stores an object *SessionData* that includes the data saved by all three components. This object is loaded and saved automatically by the BOA framework. A component can easily access the data it saved by using the *loadData* method. A component can at each moment during the negotiation update the saved information by using the *storeData* method, although we recommend updating the information at the end of the negotiation by using the the *endSession* method. The *endSession* method of each method is automatically called at the end of the negotiation to inform the component of the result obtained and should be used to update the *SessionData* object before it is automatically stored.

7.6 Advanced: Converting a BOA Agent to an Agent

To convert a BOA agent to a normal agent you have to create a class that extends *BOA agent* and override the *agentSetup* method. Below is an example of a BOA agent wrapped as a normal agent.

```
public class SimpleBOAagent extends BOAagent{

    @Override
    public void agentSetup() {
        OpponentModel om = new FrequencyModel(negotiationSession, 0.2, 1);
        OMStrategy oms = new NullStrategy(negotiationSession);
        OfferingStrategy offering  = new TimeDependent_Offering(negotiationSession,
                                om, oms, 0.2, 0, 1, 0);
        AcceptanceStrategy ac = new AC_Next(negotiationSession, offering, 1, 0);
        setDecoupledComponents(ac, offering, om, oms);
    }

    @Override
    public String getName() {
        return "SimpleBOAagent";
    }
}
```

7.7 Advanced: Multi-Acceptance Criteria (MAC)

The *BOA framework* allows us to better explore a large space of negotiation strategies. MAC can be used to scale down the negotiation space, and thereby make it better computationally explorable.

As discussed in the introduction of this chapter, the acceptance condition determines solely if a bid should be accepted. This entails that it does not influence the bidding trace, except for when it is stopped. In fact, the only difference between *BOA agents* where only the acceptance condition vary, is the time of agreement (assuming that the computational cost of the acceptance conditions are negligible).

Given this property, multiple acceptance criteria can be tested in parallel during the same negotiation trace. In practice, more than 50 variants of a simple acceptance condition as for example $\mathbf{AC}_{next}$ can be tested in the same negotiation at a negligible computational cost.

To create a multi-acceptance condition component you first need to extend the class *Mulit Acceptance Condition*, this gives access to the ACList which is a list of acceptance conditions to be tested in parallel. Furthermore, the method *isMac* should be overwritten to return *true* and the name of the components in the repository should be *Multi Acceptance Criteria*. An acceptance can be added to the MAC by appending it to the AClist as sown below.

```
public class AC_MAC extends Multi_AcceptanceCondition {
    @Override
    public void init(NegotiationSession negoSession, OfferingStrategy strat,
            OpponentModel opponentModel, HashMap<String, Double> parameters)
            throws Exception {
        this.negotiationSession = negoSession;
        this.offeringStrategy = strat;
        outcomes = new ArrayList<OutcomeTuple> ();
        ACList = new ArrayList<AcceptanceStrategy>();
        for (int e = 0; e < 5; e++) {
            ACList.add(new AC_Next(negotiationSession, offeringStrategy, 1,
                                                    e * 0.01));
        }
    }
}
```

8 Conclusion

This concludes the manual of GENIUS. If you experience problems or have suggestions on how to improve GENIUS, please send them to `ai@mmi.tudelft.nl`.

GENIUS is actively used in academic research. If you want to cite GENIUS in your paper, please refer to [**?**].

References

[1] Raz Lin, Sarit Kraus, Tim Baarslag, Dmytro Tykhonov, Koen V. Hindriks, and Catholijn M. Jonker. Genius: An integrated environment for supporting the design of generic automated negotiators. *Computational Intelligence*, 2012.