

Negotiation User Guide

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

March 5, 2018

Abstract

GENIUS [3] 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	4
1.1	Negotiation Objects	4
1.2	Optimality of a Bid	4
1.3	Negotiation Protocol	5
1.4	Reservation Value	5
1.5	Time Pressure	6
2	Protocols	6
2.1	Stacked Alternating Offers Protocol	6
2.2	Alternating Multiple Offers Protocol	6
2.3	Alternating Majority Consensus Protocol	7
2.4	Simple Mediator Based Protocol	7
2.5	Mediator Feedback Based Protocol	7
2.6	Beyond the Protocol	7
3	Running GENIUS	8
4	Scenario Creation	8
4.1	Basic GUI Components	8
4.2	Creating a Domain	8
4.3	Creating a Preference Profile	9
5	Running Negotiations	10
5.1	Running a Multi-Party Negotiation Session	10
5.2	Running a Multi-Party Tournament	11
5.2.1	Bilateral special options	12
5.3	Running from the command line	12
5.3.1	Prepare the XML settings file	12
5.3.2	Run the tournament	14
5.4	Tournament Session Generation	14
5.4.1	Multilateral generation	14
5.4.2	Bilateral generation	14
6	Quality Measures in Genius	15
6.1	Overview of Quality Measures in the Standard Log	15
6.1.1	Standard Measures	15
6.1.2	Detailed Measures	15
6.2	Overview of Quality Measures in the Tournament Log	16
6.3	Analyzing Logs using Excel	16
7	Setting up Java and IDE	17
8	Creating a Negotiation Agent	17
8.1	Receiving the Opponent's Action	17
8.2	Choosing an Action	18
8.3	General properties	18
8.4	Overview of Classes	19
8.5	Compiling an Agent	19
8.6	Loading an Agent	19
9	Creating a BOA Agent	20
9.1	Components of the BOA Framework	20
9.2	Create "Multi"lateral BOA Party	21
9.3	Bi-Lateral BOA	21
9.4	Creating New Components	23
9.4.1	Parameters	23
9.4.2	Creating a Bidding Strategy	23
9.4.3	Creating an Acceptance Condition	24
9.4.4	Creating an Opponent Model	24
9.4.5	Creating an Opponent Model Strategy	24
9.5	Compiling BOA Components	24
9.6	Adding a Component to the BOA Repository	25
9.7	Creating a ANAC2013 BOA Agent	25
9.8	Advanced: Converting a BOA Agent to an Agent	25
9.9	Advanced: Multi-Acceptance Criteria (MAC)	26

10 Creating a (Multi Party) Negotiation Agent	26
10.1 Compiling a NegotiationParty	26
10.1.1 Multiparty example	27
10.1.2 Storage example	27
10.2 Using third party libraries	27
10.3 Loading a NegotiationParty	27
10.3.1 loading with the GUI	28
10.3.2 manual loading	28
11 Conclusion	28
12 Appendix	29
12.1 Using Maven	29
12.2 Debugging	29

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 Objects

Parties participating in a negotiation interact in a domain. The domain specifies the possible bids. The parties all have their own preferences, which is reflected in their profile. Figure 1 shows a picture of a domain that describes the issues in the negotiation.

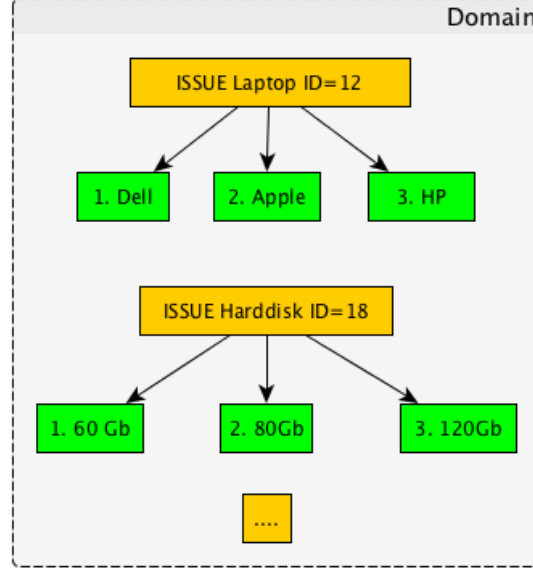


Figure 1: An example domain for laptop negotiation. Issues are orange, values are green

The *Domain* describes which issues are the subject of the negotiation and which values an issue can attain. A domain contains n issues: $D = (I_1, \dots, I_n)$. Each issue i consists of k values: $I_i = (v_1^i, \dots, v_k^i)$. Combining these concepts, an agent can formulate a *Bid*: a mapping from each issue to a chosen value (denoted by c), $b = (v_c^1, \dots, v_c^n)$.

To give an example, in the laptop domain the issues are “laptop”, “harddisk” and “monitor”. In this 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*. A valid bid in the laptop domain is a Dell laptop with 80 Gb and a 17’ inch monitor.

The *Utility Space* specifies the preferences of the bids for an agent using an evaluator. It is basically just a function that maps bids into a real number in the range $[0,1]$ where 0 is the minimum utility and 1 is the maximum utility of a bid.

A common form of the Utility space is the *Additive Utility Space*. Such a space is additive because each of the issues in the domain have their own utility of their own. For instance, we like Apple with 0.7 and Dell with 0.4, completely independent of how much memory the computer has. Figure 2 shows a picture of a utility space for the example domain that we gave above.

In an additive space 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. The utility is the weighted sum of the scaled evaluation values.

$$U(v_c^1, \dots, v_c^n) = \sum_{i=1}^n w_i \frac{\text{eval}(v_c^i)}{\max(\text{eval}(I_i))} \quad (1)$$

1.2 Optimality of a Bid

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.

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

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 3 for which there is no other bid at the top right is optimal. This type of optimality is

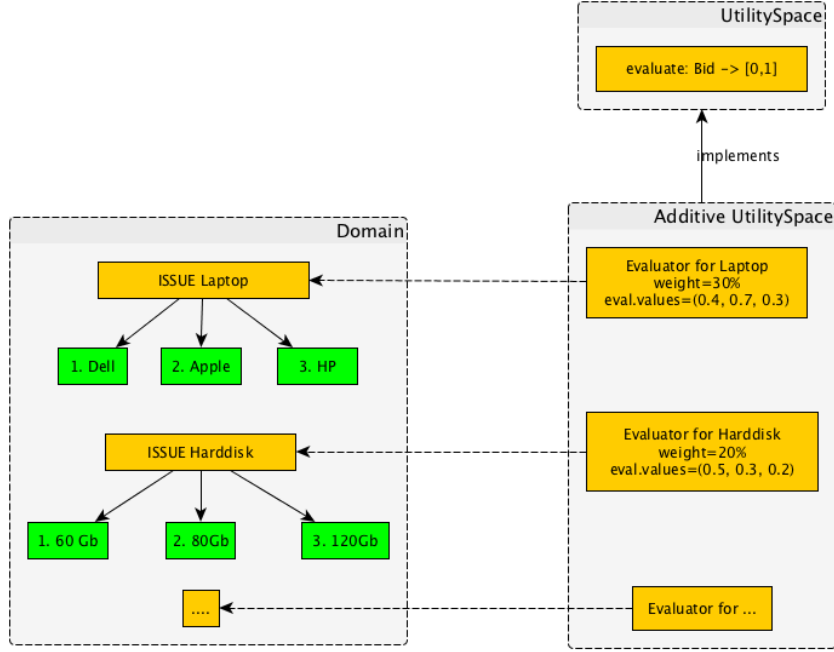


Figure 2: An example additive utility space for the laptop domain.

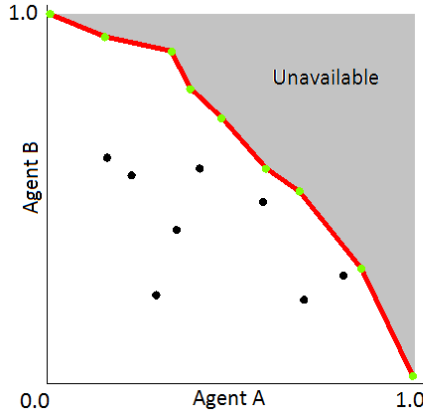


Figure 3: A point indicates the utility for both agents of a bid. The red line is the Pareto optimal frontier.

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 is that agents can 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 an *opponent 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. GENIUS provides a number of components that can estimate an opponent model.

1.3 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. GENIUS supports multiple protocols. These are discussed in detail in section 2.

1.4 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.

1.5 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 default 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:

$$U_D^t(\omega) = U(\omega) \cdot d^t \quad (2)$$

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.

2 Protocols

This section describes the various negotiation protocols. The protocol determines the overall order of actions during a negotiation. This section focuses on the MultiParty protocols as these have been properly developed. There is also a protocol class for the bilateral negotiation, but this is basically a hard coded Stacked Alternating Offers Protocol and not further developed.

The (Multilateral) protocol describes if the negotiation is finished, what the agreement is, which actions can be done in the next round. Briefly, to run a session the system checks with the protocol if the negotiation is already finished, and if not which calls need to be made to the parties (both `chooseAction` and `receiveMessage`). We recommend checking the javadoc of `MultilateralProtocol` for up-to-date detail information and how the protocol is used by the system to run sessions.

The Multilateral protocol uses the notion of rounds and turns to describe the negotiation layout. A round is a part of the negotiation where all participants get a turn to respond to the current state of the negotiation. A turn refers to the opportunity of one party to make a response to the current state of the negotiation.

If an agent violates the protocol – for instance by sending an action that is not one of the allowed ones, or by crashing, the negotiation ends and the outcome usually is ‘no agreement’ for all parties. In bilateral negotiation we have a special case then: the agent’s utility is set to its reservation value, whereas the opponent is awarded the utility of the last offer.

All protocols are found in the package `negotiator.protocol` and have the names matching the subsections below.

2.1 Stacked Alternating Offers Protocol

According to this protocol [1], all of the participants around the table get a turn per round. Turns are taken clockwise around the table. One of the negotiating parties starts the negotiation with an offer that is observed by all others immediately. Whenever an offer is made, the next party in line gets a call to `receiveMessage` containing the bid, followed by a call to `chooseAction` from which it can return the following actions:

- Accept the offer (not available the very first turn).
- send an Offer to make a counter offer (thus rejecting and overriding the previous offer, if there was any)
- send an EndNegotiation and ending the negotiation without any agreement.

This protocol is the default protocol for Parties (as returned by `getProtocol()`).

2.2 Alternating Multiple Offers Protocol

According to this protocol [1], all agents have a bid from all agents available to them, before they vote on these bids. This implemented in the following way: The protocol has a bidding phase followed by voting phases. In the bidding phase all participants put their offer on the table. These offers appear to all agents through `receiveMessage()` in a specific order. In the voting phases all participants vote on all of the bids on the negotiation table, in the same order as received. For each offer, the agent `chooseAction()` is called. If one of the bids on the negotiation table is accepted by all of the parties, then the negotiation ends with this bid.

In each even round (we start in round 0), each party gets only one turn for an OfferForVoting.

In each odd round there are N voting turns for each party (N being the number of offers), one for each offer in order of reception. these are the available options:

- Accept the offer
- Reject the offer

2.3 Alternating Majority Consensus Protocol

This protocol is essentially equal to the Alternating Multiple Offers Protocol, but now an offer the protocol keeps track of the acceptable offer that got most accepts. Initially, this may be the first offer that got one accept. After a number of rounds, some offers receive multiple accepts and these then become the new acceptable offer.

If an offer is accepted by all parties, the negotiation ends. Otherwise, the negotiation continues (unless the deadline is reached). If the deadline is reached, the acceptable offer becomes the agreement.

2.4 Simple Mediator Based Protocol

In this protocol, the parties do not hear the other parties directly. Instead, they only hear the mediator and the mediator hears the bids of all the parties. The mediator determines which bid will be voted on, collects the votes and determines the outcome. The mediator is just another NegotiationParty, but it extends Mediator.

The protocol requires that exactly one party is a Mediator. The GENIUS GUI enforces this presence of a Mediator. When you run a negotiation from the command line you have to ensure the presence of a single Mediator.

This protocol uses the following turns in every round:

1. Mediator proposes an OfferForVoting
2. The other parties (not the mediator) place a VoteForOfferAcceptance on the OfferForVoting
3. The mediator makes a InformVotingResult that informs all parties about the outcome of this round.

With this protocol, the last InformVotingResult with an accept determines the current outcome.

As mentioned, you have to provide one mediator. There is the following options

- RandomFlippingMediator. This mediator generates random bids until all agents accept. Then, it randomly flips one issue of the current offer to generate a new offer. It keeps going until the deadline is reached.
- FixedOrderFlippingMediator. This mediator behaves exactly like the RandomFlippingMediator, except that it uses a fixed-seed Random generator for every run. This makes it easier for testing.

2.5 Mediator Feedback Based Protocol

Like the Simple Mediator Based Protocol, the parties do not hear the other parties directly. Instead, they only hear the mediator and the mediator hears the bids of all the parties. The mediator determines which bid will be voted on, collects the votes and determines the outcome. The mediator is just another NegotiationParty, but it extends Mediator.

The mediator generates its first bid randomly and sends it to the negotiating agents. After each bid, each party compares the mediator's new bid with his previous bid and gives feedback ('better', 'worse' or 'same') to the mediator. For its further bids, the mediator updates the previous bid, hopefully working towards some optimum. The negotiation runs on until the deadline (unless some party crashes). This protocol is explained in detail in [2].

This protocol uses the following turns in every round:

1. Mediator proposes an OfferForFeedback.
2. The other parties (not the mediator) place a GiveFeedback, indicating whether the last bid placed by the mediator is better or worse than the previous bid.

The accepted bid is the last bid that was not receiving a 'worse' vote.

2.6 Beyond the Protocol

This section outlines the procedures surrounding running a negotiation outside the protocol.

Before the protocol can be started, the parties have to be loaded and initialized. During initialization, the party's persistent data may have to be loaded from a file. If the persistent data can not be read, a default empty dataset is created for the agent. Then the party's init code is called to set up the agent. All the time spent in this initialization phase is already being subtracted from the total available negotiation time.

After the protocol has been completed, the protocol is called a last time to determine the final outcome. The parties are called to inform them that the negotiation ended, and what the outcome was. This happens even when agents crashed or did illegal actions. The negotiation has already finished, so these calls are not weighing in on the total negotiation time. Instead, these calls are typically limited to 1 second.

Finally, if the agent has modified the persistent data, this data needs to be saved. Again, this action is limited to a 1 second duration.

Errors surrounding these out-of-protocol procedures are not part of the negotiation itself and therefore logged and handled separately. These errors are printed only to the console/terminal ¹, and only from the single session runner.

¹To see the console output, run from Eclipse or start up Genius from a separate terminal.

3 Running GENIUS

GENIUS can run on any machine running Java 8 or higher, including Windows, OSX, Solaris and Linux distributions. Please report any bugs found to negotiation@ii.tudelft.nl.

To install the environment, the file `genius-XXX.zip` can be downloaded from <http://ii.tudelft.nl/genius/?q=article/releases>. Unzip the file at a convenient location on your machine. This will result in a package called “genius-XXX” which contains the following files:

- a `userguide.pdf` which is this document.
- `genius-XXX.jar`, the negotiation simulator;
- a few example folders, containing ready-to-compile agents and components.
- a `multilateraltournament.xml` example file

When you run GENIUS (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 `genius-XXX.jar` and enter `java -jar genius-XXX.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 genius-XXX.jar`.

4 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.

4.1 Basic GUI Components

Start GENIUS by following the instructions in the previous section. After starting the simulator a screen similar to Figure 4 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.

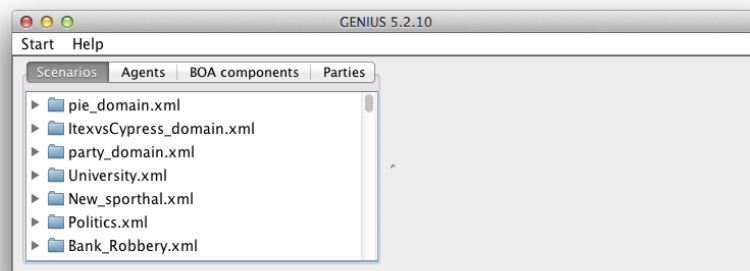


Figure 4: GENIUS right after start-up. The left half is the components panel, the right half the status panel.

4.2 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 window similar to Figure 5 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 6.

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 6 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.

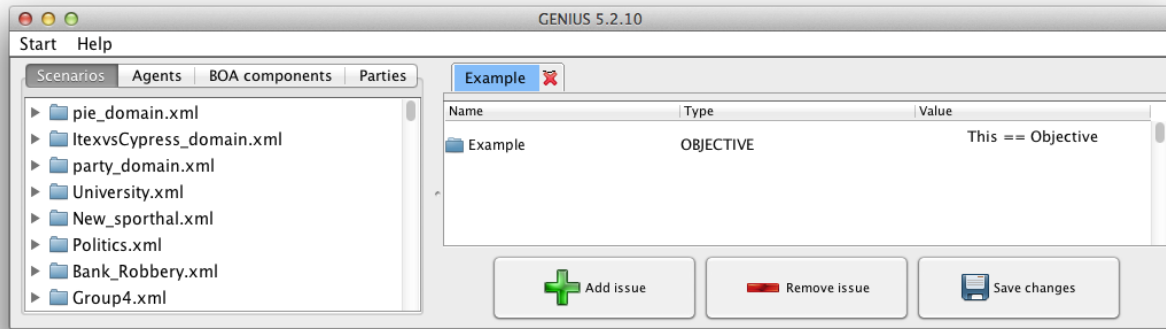


Figure 5: GENIUS after creating a new Example domain.

Instead of a discrete issue, we can also add an integer issue as shown in Figure 7. 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 [500, 700]. Next, when creating a preference profile we need to specify the utility of the lowest possible value (500) and the highest value (700). 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.

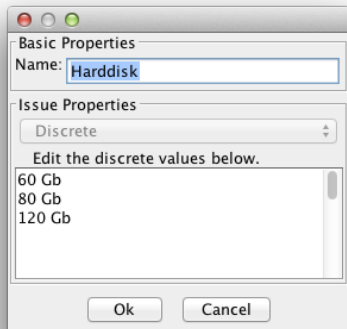


Figure 6: Creating a discrete issue.

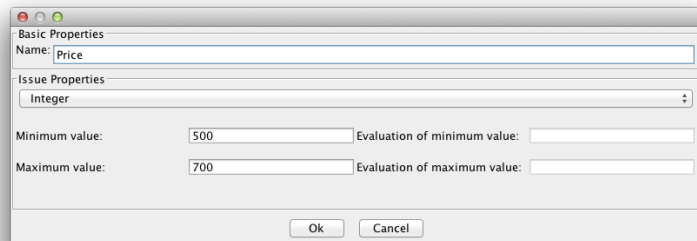


Figure 7: Creating an integer issue.

4.3 Creating a Preference Profile

Now that we created a domain, the next step is to add a set of preference profiles. Make sure that your domain is correct before proceeding, as **the domain can not be changed when it contains 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 8.

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 6 or Figure 7 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

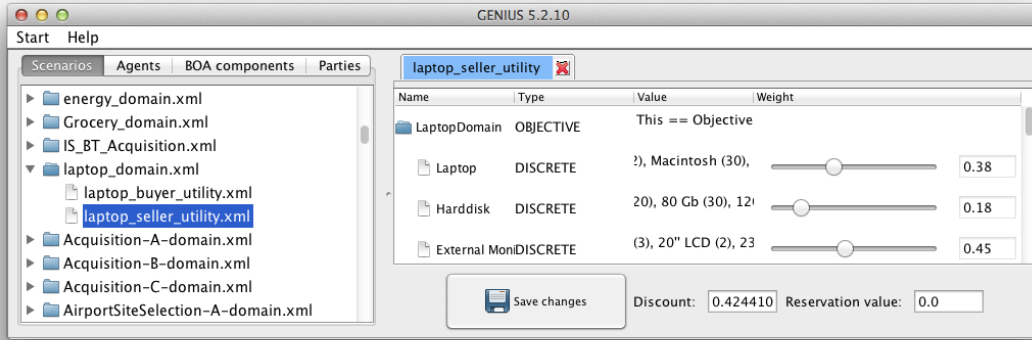


Figure 8: GENIUS after creating a new utility space.

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.

5 Running Negotiations

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

- **Multi-Party Negotiation.** A single negotiation session in which a number of agents (not necessarily 2) compete.
- **Multi-Party Tournament.** A tournament of multiparty sessions.

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.

5.1 Running a Multi-Party Negotiation Session

To run a negotiation session select the menu “Start” and then “Multi-Party Negotiation”. This opens a window similar to Figure 9.

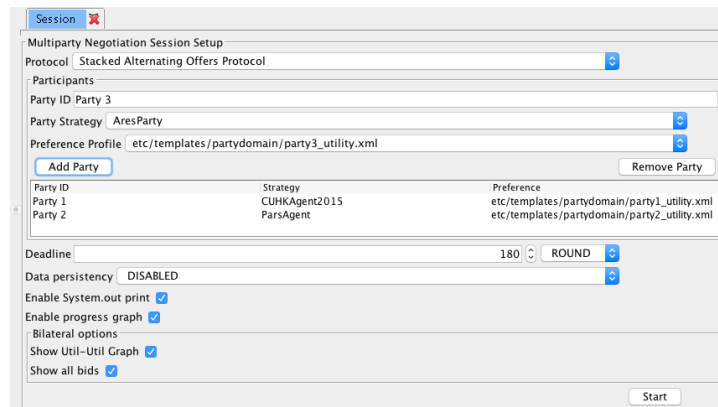


Figure 9: A multi-party negotiation session.

The following parameters need to be specified to run a negotiation:

- **Negotiation protocol.** The set of available protocols. See Chapter 2.
- **Mediator.** The mediator ID and strategy that is to be used for this session. This is only visible if the protocol uses a mediator.
- **Participant Information.** The information (ID, strategy, profile) for the a party in the session. This information is copied into the table of participants when you click "Add Party".
- **A table with participants.** This table shows all currently added participants. You can add a party by setting the participant information above, and then clicking "Add Party". You can remove a party by selecting the party to remove in the table, and then clicking "Remove Party".
- **Deadline.** The deadline to use. Can be "Round" or "Time". This determines the maximum duration of the session.
- **Data Persistency.** What kind of persistent data is available to the parties. The options are discussed in section 5.4.
- **Enable System.out print.** If disabled, all system.out.print is suppressed during the negotiation. This is useful if for instance agents are flooding the output console, slowing down the system.
- **Enable progress graph.** If enabled (default), a progress chart is shown during the negotiation. You can disable this e.g. if the drawing is slowing down the system.
- **Bilateral options** These appear only if you have exactly 2 parties added. The sub-options of this panel are
 - **Show Util-Util Graph.** If enabled, the progress panel will show a graph where the utilities of the 2 parties are set along the X and Y axes. Also, the pareto frontier and nash point are shown in this graph. If disabled, it will show the default: a graph where the utilities of all parties are along the Y axis, and the time along the X axis.
 - **Show all bids.** If enabled, and if 'Show Util-Util Graph' is enabled, this will show all the possible bids in the Util-Util graph.

The negotiation is started when you press the start button. The tab contents will change to a progress overview panel showing you the results of the negotiation (Figure 10 and Figure 11). The results are also stored in a log file. These results can be easily analyzed by importing them into Excel (cf. Section 6.3)

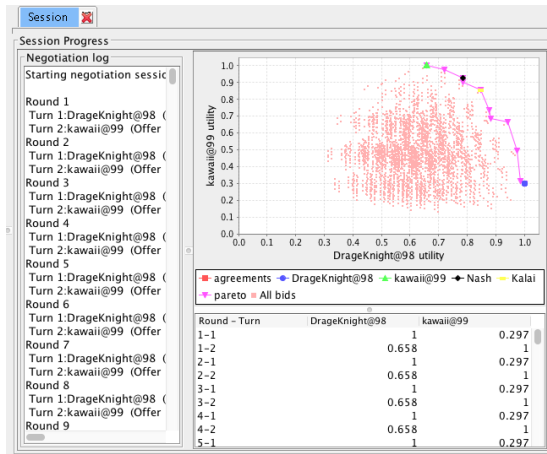


Figure 10: Bilateral progress panel.

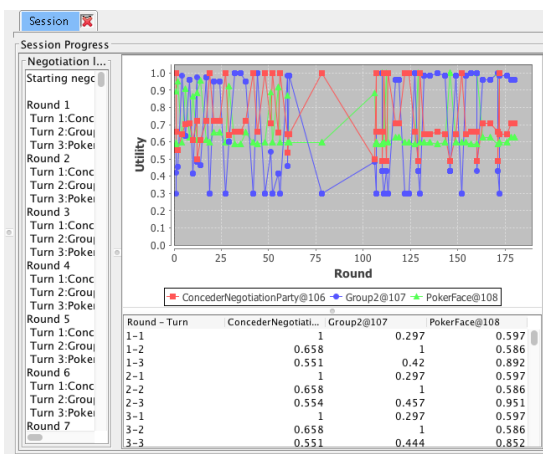


Figure 11: Multilateral progress.

5.2 Running a Multi-Party Tournament

A multi-party tournament is a set of multi-party sessions. To prepare a multi-party tournament, select "Start" and then "Multi-Party Tournament".

The Tournament tab will appear similar to Figure 12. This panel shows a set of tournament options. The detailed meaning of all these settings is explained in 5.4.

- **Protocol.** The protocol to use for each session.
- **Deadline.** The limits on time and number of rounds for each session.
- **Number of tournaments.** The number of times the entire tournament will be run.
- **Agents per Session.** The number of agents N to use for each session.
- **Agent Repetition.** whether to draw parties with or without return.
- **Randomize session order.** whether to randomize the session order

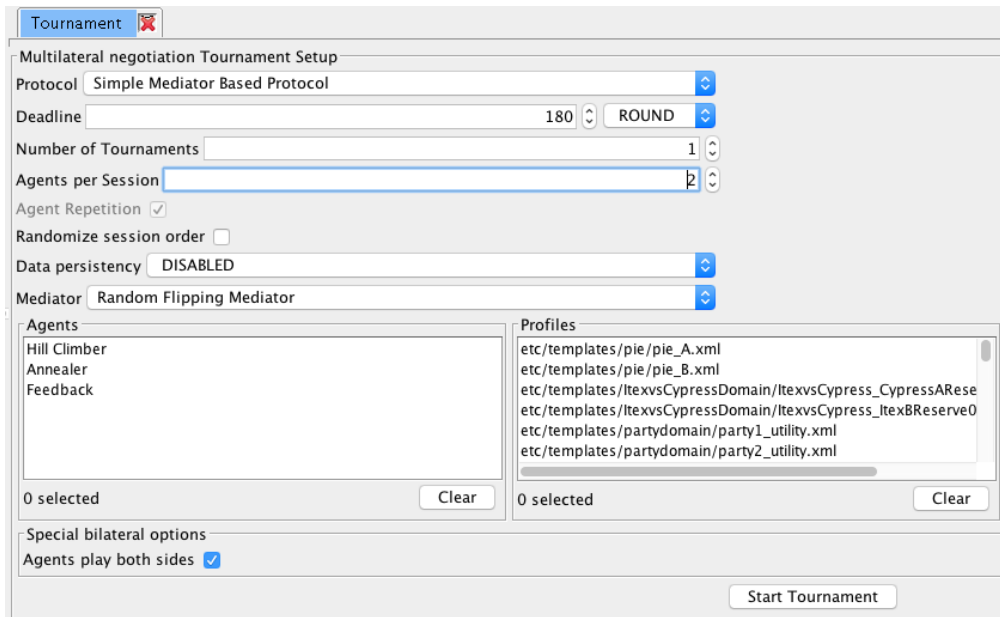


Figure 12: Multi-Party Tournament

- **Data persistency.** The type of persistent data available to the parties. Same options as in section 5.1.
- **Mediator.** The mediator to use. This option is visible only if the selected protocol needs a mediator.
- **Agents.** The pool of agents to draw from. Click or drag in the agents area to (de)select agents. Click "Clear" to clear the pool.
- **Profiles.** The profiles pool. Click or drag in the profiles area to (de)select agents. Click "Clear" to clear the pool.
- **Special bilateral options.** These options appear only if Agents per session is set to 2 and is discussed in below .

5.2.1 Bilateral special options

If you have set 'Agents per session' to 2, and deselect 'Agent play both sides', you get an additional panel where you can select different Agents and Profiles for the B side of the 2-sided negotiation as in Figure 13.

After you click "Start Tournament", the tournament starts. The panel then is swapped for a tournament progress panel (Figure 14). In the top there is a progress bar showing the total number of sessions and the current session. The table shows all session results. The table is also saved to a .csv log file in the log directory.

The results of the tournament are shown on screen and also stored in a log file. These results can be easily analyzed by importing them into Excel (cf. Section 6.3)

5.3 Running from the command line

You can run a multi-party tournament from the command line, as follows.

1. Prepare an xml file that describes the settings for the tournament
2. Run the command runner and give it the prepared file

5.3.1 Prepare the XML settings file

The first step is to create an xml file containing the values needed for session generation (Section 5.4). Make a copy of the `multilateraltournament.xml` file inside your genius directory and edit it (with a plain text editor). Inside the `<tournaments>` element you will find a number of `<tournament>` elements. Each of these `<tournament>` elements defines a complete tournament so you can run multiple tournaments using one xml file.

The contents of each `<tournament>` element is as follows. The meaning of the fields is detailed in section 5.4.

- **protocolItem.** Contains the protocol to use, in the form of a protocolItem.
- **deadline.** the Deadline value.
- **repeats.** the repeats value.
- **persistentDataType.** The type of the persistent data.
- **numberOfPartiesPerSession.** the Parties per session value.
- **repetitionAllowed.** the value for the Party Repetition.

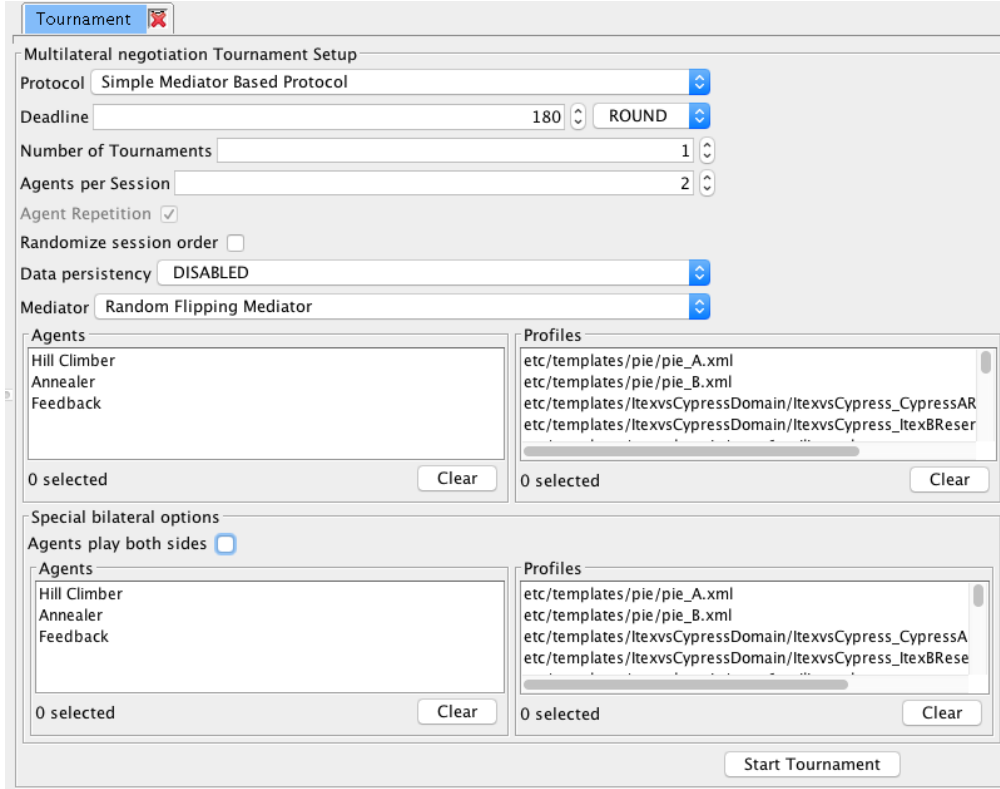


Figure 13: Multi-Party Bilateral Tournament

Tournament

48/48

Run time (s)	Round	Exception	deadline	Agreement	Discounted	#agreeing	min.util.	max.util.	Dist. to Par...	Dist. to Nash	Social Welf...	Agent 1	Agent 2
0.327	150		180rounds	Yes	No	2	0.54928	0.95790	0.08267	0.32926	1.50718	AgentH@0	AresPa
0.286	124		180rounds	Yes	No	2	0.67185	0.98218	0.00000	0.22569	1.65403	SENGOKU...	AresPa
0.236	129		180rounds	Yes	No	2	0.67185	0.98218	0.00000	0.22569	1.65403	AgentW@4	AresPa
0.020	12		180rounds	Yes	No	2	0.67162	0.96285	0.01934	0.21662	1.63446	AgentX@6	AresPa
0.214	117		180rounds	Yes	No	2	0.70238	0.94630	0.05874	0.25032	1.64868	AgentH@8	AresPa
0.021	11		180rounds	Yes	No	2	0.91667	0.95792	0.00000	0.04319	1.87458	SENGOKU...	AresPa
1.740	74		180rounds	Yes	No	2	0.91667	0.95792	0.00000	0.04319	1.87458	AgentW@12	AresPa
0.006	3		180rounds	Yes	No	2	0.72619	1.00000	0.00000	0.23572	1.72619	AgentX@14	AresPa
0.019	12		180rounds	Yes	No	2	0.96535	0.96535	0.04900	0.04900	1.93070	AgentH@16	AresPa
0.004	3		180rounds	Yes	No	2	1.00000	1.00000	0.00000	0.00000	2.00000	SENGOKU...	AresPa
0.005	3		180rounds	Yes	No	2	1.00000	1.00000	0.00000	0.00000	2.00000	AgentW@20	AresPa
0.005	3		180rounds	Yes	No	2	1.00000	1.00000	0.00000	0.00000	2.00000	AgentX@22	AresPa
0.271	172		180rounds	Yes	No	2	0.47813	1.00000	0.00000	0.40083	1.47813	AgentH@24	AresPa

Figure 14: Tournament Progress panel

- **partyRepItems**. This element contains a number of `<item>` elements. Each of these party items contains a description of a party as discussed below.
- **mediator**. the mediator, if needed. This is similar in contents to a party item discussed below.
- **partyProfileItems**. This element contains a number of items. There must be at least as much as `numberOfNon-MediatorsPerSession`.

We have a number of items:

- A profile item : contains
 - **url** that contains the description of that party profile. These URIs point to files and therefore are of the form `file:path/to/file.xml`
- A party item (and mediator) contains:
 - **classPath** the java.party.class.path to the main class. That class must implement the `NegotiationParty` interface
 - **properties** can contain a number of `<property>` nodes with these values
 - * **isMediator**: this property indicates the party item is a mediator. If not set, the party will be run as a normal party instead of a mediator, which will probably cause protocol violations
- protocol item. This item contains the protocol information:
 - **hasMediator** which is true iff protocol requires mediator

- **description** a one-line textual description of the mediator
- **classPath** the java full.class.path of the protocol class
- **protocolName** a brief protocol name

The tournament will consist of sessions created creating all permutations of `<numberOfNonMediatorsPerSession>` from the partyRepItems (with or without reuse, depending on **repetitionAllowed**). The randomization also is applied to the profile items.

5.3.2 Run the tournament

To run the tournament, open a terminal/console and change the working directory to the genius directory. Then enter this command (where yourfile.xml is the name of the file you just edited and XXX the version of genius that you use):

```
java -cp genius-XXX-jar-with-dependencies.jar genius.cli.Runner yourfile.xml
```

Press return if the app prompts you for the log file location to log to the default `logs/...csv` file.

5.4 Tournament Session Generation

Instead of manually setting all the setting, a tournament generates the exact session settings from the tournament settings. These settings are specified either in the user interface settings, or in an XML file. The parameters are:

- **Protocol** The protocol value is used for all sessions. See section 2.
- **Mediator** The mediator to use for all sessions (ignored if the protocol does not need a mediator)
- **Deadline** The deadline is used for all sessions. A deadline contains two values:
 - **value**. This is the maximum value determining the deadline. Must be an integer ≥ 1 .
 - **type**. Can be either *ROUND* or *TIME*. If *ROUND*, the value is the number of rounds. If *TIME*, value is a time in seconds.
- **Data persistency**. The type of persistent data available to the parties. The next time an agent of the same class and same profile runs in a tournament, it will receive the previously stored data. The options are
 - **Disabled**. Parties do not receive any persistent data. This is the default.
 - **Serializable**. Parties can save anything serializable in the *PersistentDataContainer*.
 - **Standard**. Parties receive a prepared, read only StandardInfo object inside the *PersistentDataContainer*.
- **repeats** This is also called 'number of tournaments' and determines the number of times a complete tournament will be run.
- **Randomize Session Order** Whether all generated sessions within a tournament must be randomized.
- **Parties per session** The number of agents to draw for each session. This excludes a possible mediator.
- **Party Repetition** true if agents are to be drawn from the agents pool with return, false if they are to be drawn without return.
- **Parties and Profile pool for side A** A list from which parties and profiles will be drawn
- **Parties and Profile pool for side B** Another list of parties and profiles. Only used with bilateral generation (see below).

The tournament generation works as follows.

If there are exactly 2 agents per session and the agents and profiles for side B have been set, then bilateral generation is used. Otherwise, multilateral generation is used. This generation method creates an ordered list of sessions for 1 tournament. If the 'Randomize Session Order' is set, the list is randomized. All sessions use the same protocol, mediator, deadline and data persistency. This generation is called repeatedly, as set in 'repeats', and all generated session lists are accumulated in a big session list. This is the final result of the tournament generation.

5.4.1 Multilateral generation

In multilateral generation, all possible combinations of parties and profiles (using pool A) are generated as follows. the indicated number of parties per session N are drawn from agent pool A, with or without return as specified in 'Party Repetition'. Also, N profile items are drawn, ordered without return, from the profiles pool. These two lists are then paired into groups of N party-profile pairs.

5.4.2 Bilateral generation

In bilateral generation, first a set of participants P of all combinations of 1 party and 1 profile are drawn from the side A pool. Similarly a set of participants Q is drawn for the B pool. Then, the sessions set consists of all combinations of one participant from P and another participant from Q .

6 Quality Measures in Genius

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

There are three types of logs used in GENIUS : the standard log, the tournament log and the tournament statistics 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.
- The Tournament statistics log is logged only for the Multiparty Tournament. The file is written in the `logs/` directory when the tournament finishes. It has the same filename as the tournament log, but with an additions "Stats". It contains statistics for each agent in the tournament.

First, Section 6.1 discusses the measures incorporated in the standard log. Next, Section 6.2 details the tournament log. Finally, Section 6.3 discusses how Excel can be used to analyze logs.

6.1 Overview of Quality Measures in the Standard Log

Since version 4.0, GENIUS 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.

6.1.1 Standard Measures

The standard measures are the measures which are enabled by default and cannot be disabled. Table 1 provides an overview of all default quality measures.

Attribute	Description
acceptance_strategy	The acceptance strategy of a BOA agent (see Section 9).
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.
bestDiscountedAcceptableBid	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 9).
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 9).
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.

6.1.2 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 6.2.

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.

6.2 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.

6.3 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 15), 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.

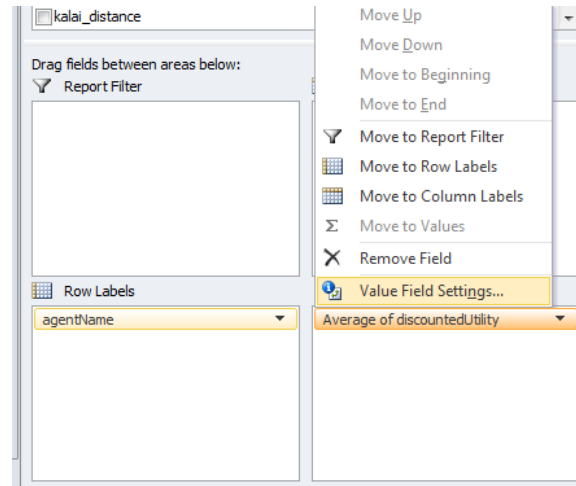


Figure 15: Configuration required to summarize the discounted utility of each agent.

7 Setting up Java and IDE

We assume that you are familiar with programming in Java. In case you are not familiar with Java, please consult the following tutorial: <http://www.oracle.com/technetwork/java/javase/documentation/index.html>

The recommended way to develop an agent is to create a new project in for example Eclipse or Netbeans. Please refer to Appendix 12 where the configuration procedure is explained in detail.

In the examples we will use manual compilation using maven to avoid the need to discuss IDE peculiarities.

8 Creating a Negotiation Agent

This section discusses how to create a basic negotiation agent in Java. 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 9. The main advantage of a BOA agent is that existing components can be reused, allowing for easier agent development.

Finally, to create an agent create a new class and extend the *negotiator.Agent* class. Table 3 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 <code>getTime()</code> .	
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.

8.1 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;
}

```

8.2 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} :

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

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 16 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.

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.

8.3 General properties

Some agents have restrictions and can not be used in certain situations. The agent indicates its capabilities through the function `getSupportedNegotiationSetting()`. By default, the agent has no restrictions. If your agent has restrictions, you must override this function and return the appropriate supported settings.

For example, if your agent can only handle linear utility spaces, you should override like this

```

@Override
public SupportedNegotiationSetting getSupportedNegotiationSetting() {
    return SupportedNegotiationSetting.getLinearUtilitySpaceInstance();
}

```

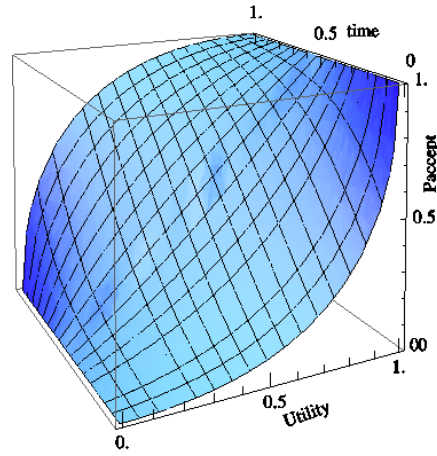


Figure 16: P_{accept} value as function of the utility and time (as a fraction of the total available time).

8.4 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.

8.5 Compiling an Agent

Compiling an agent can be done as follows (here we compile the examplepackage; modify as appropriate for your agent):

- Open a terminal
- Switch to the root directory of genius
- execute the command

```
javac -cp genius-XXX.jar -source 1.8 -target 1.8 examplepackage/ExampleAgent.java
```

You can also compile from Eclipse or Netbeans. Make sure you add the genius-XXX.jar to your class path. Please refer to the Eclipse or Netbeans documentation on how to do this. Also you can check our tutorial on how to do this from Eclipse.

8.6 Loading an Agent

The next step is to load the compiled agent in GENIUS . 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 17). 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.
- **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 where the compiled agent class is located. For built-in agents this is the class name but for user-defined agents this is the full filename of the main class of the agent (the one implementing the Agent interface). The third element specifies the *agentName*. 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.

Scenarios		Agents	BOA components
Agent Name	Description		
ANAC 2010 - IAMcrazyHaggler	ANAC 2010 - IAMcrazyHaggler		
ANAC 2010 - AgentSmith	ANAC 2010 - AgentSmith		
ANAC 2010 - Nozomi	ANAC 2010 - Nozomi		
ANAC 2011 - TheNegotiator	ANAC 2011 - TheNegotiator		
ANAC 2011 - ValueModelAgent	ANAC 2011 - ValueModelAgent		
ANAC 2011 - BRAMAgent	ANAC 2011 - BRAMAgent		
ANAC 2011 - HardHeaded	ANAC 2011 - HardHeaded		
ANAC 2011 - IAMhaggler2011	ANAC 2011 - IAMhaggler2011		
ANAC 2011 - AgentK2	ANAC 2011 - AgentK2		
ANAC 2012 - AgentLG	ANAC 2012 - AgentLG		
ANAC 2012 - AgentTMR	ANAC 2012 - AgentTMR		

Figure 17: Adding an agent using the GUI.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<repository fileName="agentrepository.xml">
  <items>
    <agentRepItems>
      <agentRepItem description="Simple Agent"
        classPath="/Users/genius/examplepackage/ExampleAgent.class"
        agentName="Simple Agent"/>
      </agentRepItems>
    </items>
  </repository>
```

9 Creating a BOA Agent

Instead of implementing your negotiating agent from scratch, you can create a BOA agent using the *BOA framework*. However, notice that the current BOA components only are suited for bi-lateral negotiation, not for general multi-lateral negotiation. 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*.

BOA components can be used both in bilateral tournaments and multilateral negotiation (both single sessions and tournaments).

9.1 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 18). 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.

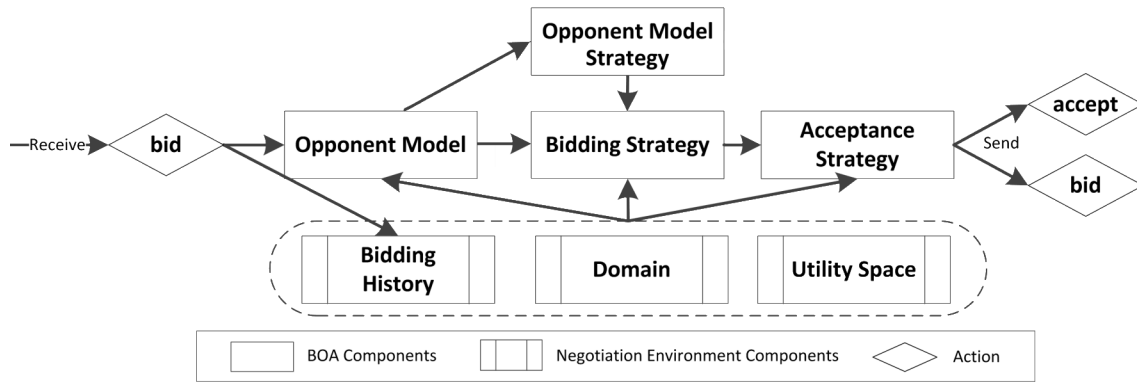


Figure 18: The BOA Framework Architecture.

9.2 Create "Multi"lateral BOA Party

For use in the multi-lateral negotiation system, the BOA agents can be edited in the "Boa Parties" repository tab (Figure 19). Right-click in the panel to add items. Select an item and right-click to remove or edit an item. These BOA parties are at this moment still a simple extension of the old bilateral system, and can *only be used in multi-lateral negotiation with exactly 2 parties per session*.

Scenarios	Agents	BOA components	Parties	Boa Parties
Name	BIDDINGSTRATE...	ACCEPTANCEST...	OPPONENTMODEL	OMSTRATEGY
test2boa	2012 - CUHKA...	2012 - CUHKA...	CUHK Frequen...	TheFawkes[]
test3	2010 - Nozomi[]	2011 - BRAMA...	Bayesian Mode...	TheFawkes[]
boatest10	2010 - IAMha...	2012 - TheNe...	Default[]	NTFT[]

Figure 19: The BOA Parties repository tab.

After you selected to add or edit a BOA party (Figure 20). Here you can select a different Bidding Strategy, Acceptance Strategy, Opponent Model and Opponent Model Strategy by selecting the appropriate strategy with the combo boxes. If the strategy has parameters, the current parameter settings are shown and the respective "Change" button enables.

Figure 20: Editing a BOA party.

If you click on the "Change" button, another panel pops up where you can edit the parameters (Figure 21). You can click directly in the table to edit values.

When you have correctly set all strategies and their parameters, you can click the "OK" button in the BOA party editor (Figure 20). Then, agents with the given name are generated, one for each permutation of the range of settings you set in the parameters. For example, if you set you want parameter m to have values 0,1 and 2 and x to have values 7 and 8, there will appear 6 new agents, with settings [0,7],[0,8],[1,7],[1,8],[2,7], and [2,8]. Be careful with this generation as it is easy to create an excessive amount of agents this way.

9.3 Bi-Lateral BOA

In this section we create a *BOA agent* by selecting its components from a list of existing components.

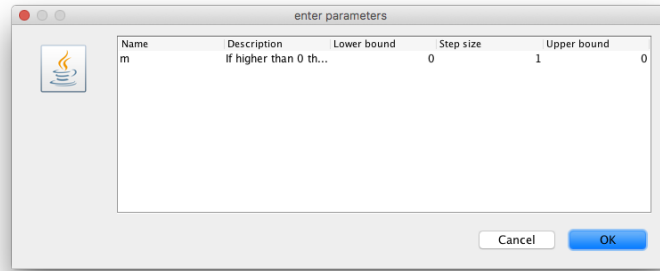


Figure 21: Editing the Parameters of a BOA party.

In the bilateral tournament runner you edit the BOA agents in the tournament settings, and the Boa Parties repository is ignored. The BOA framework GUI (see Figure 22) 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.

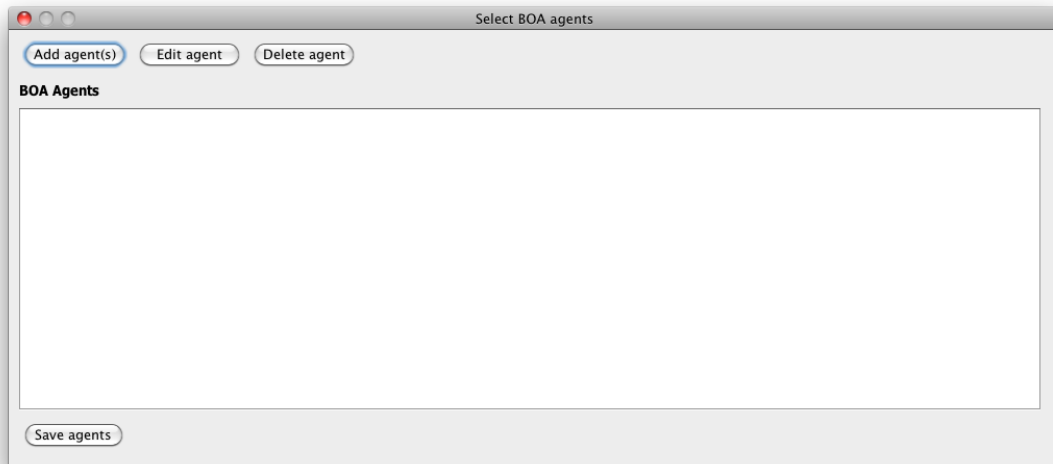


Figure 22: The BOA framework GUI.

Our goal in this section is to specify three BOA agents which are equal except for a single parameter a of their acceptance strategy.

To add the agents, click on the "Add agent(s)" button. A dialog pops up to enter the BOA agent details (Figure 23).

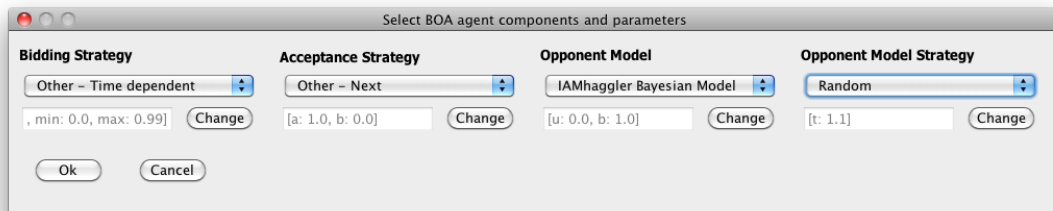


Figure 23: The BOA agent components and parameters dialog.

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 23.

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, press the "Change" button under *Acceptance Strategy* to open a window similar to Figure 24. Next, fill in the fields as shown in Figure 24. Finally, we select "Add agent(s)" to create the three agents. Press "Save agents" to save the new BOA agents for the tournament. 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.

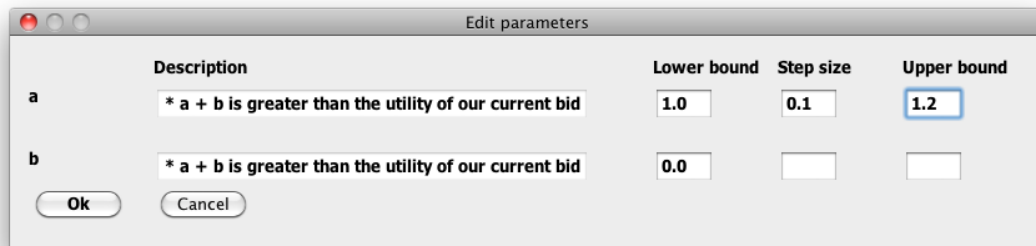


Figure 24: Adding a parameter.

9.4 Creating New Components

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

9.4.1 Parameters

All BOA components have the same mechanism to be tuned with parameters. They should have no constructor : the default empty constructor will be called. They initialize through a call to `init()`.

The parameters and their default parameters are indicated by the component by overriding the `getParameters()` function. This function should return a set of *BAOparameter* objects, each parameter having a unique name, description and default value.

```
public Set<BOAparameter> getParameterSpec()
Override this function to add parameters to the module.
```

Table 4: The `getParameters` method. Override if your component has parameters.

When the component is actually used, the actual values for the parameters (which may differ from the default) are passed to the `init` function when the component is initialized.

9.4.2 Creating a Bidding Strategy

A bidding strategy can be easily created by extending the *OfferingStrategy* class. Table 5 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 *boaexamplepackage* directory.

```
void init(NegotiationSession negotiationSession, OpponentModel opponentModel, OMStrategy
omStrategy, Map<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 5: The main methods of the bidding strategy component.

9.4.3 Creating an Acceptance Condition

This section discusses how to create an acceptance strategy class by extending the abstract class *AcceptanceStrategy*. Table 6 depicts the two methods which need to be specified.

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

Table 6: The main methods of the acceptance strategy component.

9.4.4 Creating an Opponent Model

This section discusses how to create an opponent model by extending the abstract class *OpponentModel*. Table 7 provides an overview of the main methods which need to be specified. For performance reasons it is recommended to use the *UtilitySpace* class.

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

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

9.4.5 Creating an Opponent Model Strategy

This section discusses how to create an opponent model strategy by extending the abstract class *OMStrategy*. Table 8 provides an overview of the main methods which need to be specified.

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

Table 8: The main methods of the opponent model strategy component.

9.5 Compiling BOA Components

BOA components must be compiled before they can be loaded into Genius. To compile a BOA component, do the following steps (in this example we compile the boa example components)

- Open a terminal
- Switch to the root directory of genius
- Enter the command `javac -cp genius-XXX.jar -source 1.8 -target 1.8 boaexamplepackage/*.java`

You can also compile from Eclipse or Netbeans. Make sure you add the genius-XXX.jar to your class path. Please refer to the Eclipse or Netbeans documentation on how to do this.

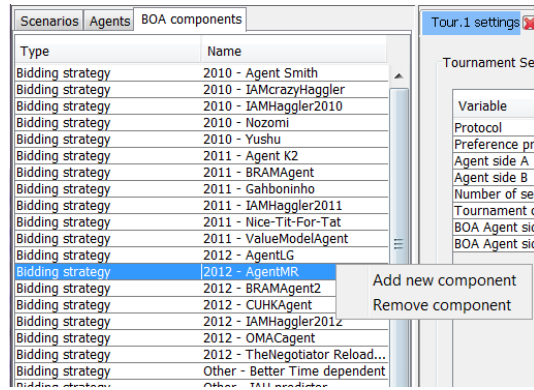


Figure 25: The BOA components window.

9.6 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, open the BOA components tab in the components window as shown in Figure 25. Right click and select “Add new component”. This results in the opening of the window shown in Figure 26.

Click on the “Open” button and select the main class file of your BOA component (the class file that implements the BOA interface). Then check the name of the component, you can change it but it has to be a unique name in the registry. Optionally add parameters. Finally, clicking “Add component” in this window adds the component to the repository.

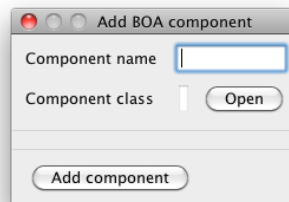


Figure 26: Loading a BOA agent.

9.7 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 *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.

9.8 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 SimpleBOAgent extends BOAgent{

    @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";
    }
}

```

9.9 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 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 *Multi 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 shown below.

Listing 1: Example code for Acceptance condition

```

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));
        }
    }
}

```

10 Creating a (Multi Party) Negotiation Agent

This section discusses how to create a multilateral negotiation agent in Java. Obviously "multi" includes 2 so you can do bilateral negotiations with these agents as well.

To implement a multi-party negotiation party, at a minimum one needs to implement a class that implements the `textttnegotiator.parties.NegotiationParty` interface (Table 9). Also your implementation must have a public default (no-argument) constructor. Please refer to the javadocs for details on the parameters.

For convenience, you can also extend the class `negotiator.parties.AbstractNegotiationParty`. This class provides convenient support functions for building your agent.

Your agent might need check the provided `AbstractUtilitySpace`, for instance if your agent supports for example only `AdditiveUtilitySpace`.

We recommend to use the javadoc included with the distribution to check the details of all the involved classes.

The functions `receiveMessage` and `chooseAction` are basically the same as described in section 8.1 and 8.2. `getProtocol` by default returns an instance of `StackedAlternatingOffersProtocol`. Your agent should override this if it works on a different protocol.

10.1 Compiling a NegotiationParty

Here we discuss two simple examples showing some aspects of a multiparty agent.

Method	description
init	Initializes the party, informing it of many negotiation details. This MUST be called exactly once, immediately after construction of any class implementing this
chooseAction	When this function is called, it is expected that the Party chooses one of the actions from the possible action list and returns an instance of the chosen action.
receiveMessage	this method is called to inform the party that another @link NegotiationParty chose an @link Action.
getDescription	a human-readable description for this party
getProtocol	the actual supported MultilateralProtocol, usually StackedAlternatingOffersProtocol.
negotiationEnded	This is called to inform the agent that the negotiation has been ended. This allows the agent to record some final conclusions about the run

Table 9: Methods of NegotiationParty. Check the javadoc for all the details

10.1.1 Multiparty example

This example party just accepts any acceptable bid with a random probability of 0.5. To compile the example agent, go to your genius project folder and use the command line to execute this compile command

```
javac -cp genius-1.0.0-jar-with-dependencies.jar -source 1.8 -target 1.8 multipartyexample/Groupn.java
```

You can run this agent against a team of parties already provided in GENIUS .

10.1.2 Storage example

Another example is demonstrating the persistent data storage. This example is showing how the storage can be used to wait a little longer every next time the party is in a negotiation.

Compile the example agent in the storageexample using

```
javac -cp genius-XXX-jar-with-dependencies.jar -source 1.8 -target 1.8 storageexample/GroupX.java
```

Then, load it into GENIUS and run it in a multiparty tournament. Set the tournament with the following settings

- number of tournaments= 20
- agents per session =2
- persistency=standard
- agent side A: GroupX, party1_utility.xml
- agent side B: Random Party, party6_utility.xml

and start the tournament and check the number of rounds till agreement: it will increase every session.

Now run another tournament with the same settings but pick select both party1_utility.xml and party2_utility.xml. Run the tournament. Now you will see that the the number of rounds till agreement goes up every other run. This is because your agent gets a different profile every other run and thus there are persistent data stores, one for each profile.

10.2 Using third party libraries

If you want to use a third party library, you will have to include all the source code of that library with your code, including all sub-dependencies. The code should be copied inside the package name of your agent, instead of using the original package name of that library (so do not use "org.apache" for instance). This is to ensure that we are really running your agent on the specific version of the library that your agent needs and to avoid version conflicts (java will run an unspecified version of the library in case of conflicts).

10.3 Loading a NegotiationParty

You need to load your custom party into the party repository in order to use it. After adding, your agent will appear in the combo boxes in the multilateral tournament runner and session runner where you can select the party to use.

You can load the new NegotiationParty into the party repository in two ways:

10.3.1 loading with the GUI

Locate the Parties repository tab in the GUI (Figure 27). Right click in this area and select "Add Party". A browser window pops up. Brows to your compiled class file that implements the `NegotiationParty` and select it. Your party will appear at the bottom of the parties repository. The `partyrepository.xml` file is automatically updated accordingly.

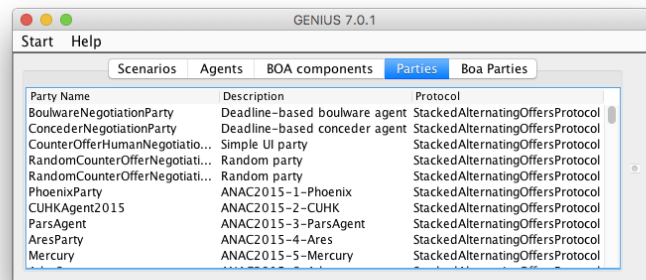


Figure 27: The parties repository.

10.3.2 manual loading

To do this manually, quit GENIUS , open the `partyrepository.xml` file and add a section like this

```
<partyRepItem classPath="class.of.your.agent" <properties/> />
```

After that you can restart GENIUS .

11 Conclusion

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

GENIUS is actively used in academic research. If you want to cite GENIUS in your paper, please refer to [3].

12 Appendix

This appendix describes how to set up Eclipse to create and debug your own agent. You can also create a maven project by just installing maven and creating a correctly formatted pom.xml, please refer to online maven tutorials for this.

12.1 Using Maven

Select in the workbench File/New Project... and select Maven/new Maven project. Select "create a simple project" and click next. Enter some group id (eg "ai2019") and some artifactID (eg "group20"). Click finish.

open the pom.xml file and insert this inside the <project> part:

```
<dependencies>
  <dependency>
    <groupId>genius</groupId>
    <artifactId>gui</artifactId>
    <version>8.0.0</version>
  </dependency>
</dependencies>

<repositories>
  <repository>
    <id>artifactory.ewi.tudelft.nl</id>
    <url>http://artifactory.ewi.tudelft.nl/artifactory/libs-release</url>
  </repository>
  <repository>
    <id>netbeans</id>
    <name>Netbeans rep</name>
    <url>http://bits.netbeans.org/maven2/</url>
  </repository>
</repositories>
```

Add your agent under the src/main/java. Eclipse now automatically downloads and includes all dependencies into your project and build your agent into target/classes.

To start Genius for debugging, do the following

1. Run/Debug Configurations...
2. Select "Java Application" and click "new" button.
3. enter `genius.Application` in the "Main Class" field.
4. Click "Debug".

NOTICE: Strictly, your agent should compile with only this dependency:

```
<dependencies>
  <dependency>
    <groupId>genius</groupId>
    <artifactId>core</artifactId>
    <version>1.0.1</version>
  </dependency>
</dependencies>
```

However, we suggest using the gui dependency for now because of convenience of debugging and running your agent. Take care that your agent does not re-use code from old agents, and use only the core APIs.

12.2 Debugging

Once you have Genius running in Eclipse, you can simply place a breakpoint in your agent and run Genius from Eclipse in debug mode.

References

- [1] Reyhan Aydogan, David Festen, Koen V. Hindriks, and Catholijn M. Jonker. Alternating offers protocols for multi-lateral negotiation. *Modern Approaches to Agent-based Complex Automated Negotiation*, 2014.
- [2] Reyhan Aydogan, Koen V. Hindriks, and Catholijn M. Jonker. Multilateral mediated negotiation protocols with feedback. In I. Marsa-Maestre, M. A. Lopez-Carmona, T. Ito, M. Zhang, Q. Bai, and K. Fujita, editors, *Novel Insights in Agent based Complex Automated Negotiation, Chapter: Multilateral Mediated Negotiation Protocols with Feedback*, chapter 3, pages 43–59. Springer, 2014.
- [3] 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.