Using the GENIUS Framework for Running Autonomous Negotiating Parties

T. Baarslag, W. Pasman, K. Hindriks, D. Tykhonov

August 20, 2018

Abstract

GENIUS [**?**] is a negotiation environment that implements an open architecture for heterogeneous negotiating parties. 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 parties, and write, compile, and run an party yourself.

Contents

1 Theory Crash Course **4**
- 1.1 Bids, Issues and Values . . . 4
- 1.2 Preference Profile, Utility Space . . . 4
  - 1.2.1 Utility spaces . . . 4
  - 1.2.2 Partially ordered profile . . . 5
- 1.3 Optimality of a Bid . . . 5
- 1.4 Negotiation Protocol . . . 5
- 1.5 Reservation Value . . . 6
- 1.6 Time Pressure . . . 6

2 Protocols **6**
- 2.1 Stacked Alternating Offers Protocol . . . 7
- 2.2 Alternating Multiple Offers Protocol . . . 7
- 2.3 Alternating Majority Consensus Protocol . . . 7
- 2.4 Simple Mediator Based Protocol . . . 7
- 2.5 Mediator Feedback Based Protocol . . . 8
- 2.6 Beyond the Protocol . . . 8

3 Install and Run Genius **8**

4 Scenario Creation **9**
- 4.1 Creating a Domain . . . 9
- 4.2 Creating an AdditiveUtilitySpace . . . 10
- 4.3 Creating an UncertainAdditiveUtilitySpace . . . 11

5 Running Negotiations **11**
- 5.1 Running a Session . . . 11
- 5.2 Running a Tournament . . . 12
  - 5.2.1 Bilateral special options . . . 13
- 5.3 Running from the command line . . . 14
  - 5.3.1 Prepare the XML settings file . . . 14
  - 5.3.2 Run the tournament . . . 15
- 5.4 Tournament Session Generation . . . 15
  - 5.4.1 Multilateral generation . . . 15
  - 5.4.2 Bilateral generation . . . 15

6 Quality Measures **16**
- 6.1 Session logs . . . 16
  - 6.1.1 Session CSV file . . . 16
  - 6.1.2 Session XML file . . . 16
- 6.2 Tournament logs . . . 17
  - 6.2.1 Tournament log.csv file . . . 17
  - 6.2.2 Tournament log.xml file . . . 18
  - 6.2.3 Tournament logStats.xml file . . . 18

7 Creating a Negotiation Party **18**
- 7.1 Example Parties . . . 18
- 7.2 Implementing NegotiationParty . . . 19
  - 7.2.1 Receiving the Opponent's Action . . . 20
  - 7.2.2 Choosing an Action . . . 20
- 7.3 Implementing a party with uncertainty . . . 21
  - 7.3.1 Overriding functions . . . 21
  - 7.3.2 Bid Ranking . . . 22
  - 7.3.3 Accessing the real utility space for debugging . . . 22
  - 7.3.4 Uncertainty agent checklist . . . 22
- 7.4 Loading a NegotiationParty . . . 22
- 7.5 Third party code . . . 23

8 Creating a BOA Party **23**
8.1 Components of the BOA Framework . . . 23
8.2 Create a BOA Party . . . 24
8.3 Creating BOA Components . . . 25
8.3.1 Set up a Workspace . . . 25
8.3.2 Add component to Genius . . . 25
8.3.3 Parameters . . . 25
8.3.4 Creating an Offering Strategy . . . 25
8.3.5 Creating an Acceptance Condition . . . 26
8.3.6 Creating an Opponent Model . . . 26
8.3.7 Creating an Opponent Model Strategy . . . 26
8.4 Advanced: Converting a BOA Party to a Party . . . 26
8.5 Advanced: Multi-Acceptance Criteria (MAC) . . . 27

9 Debugging **28**
9.1 Source code and javadocs . . . 28

10 Conclusion **28**

11 Appendix **29**
11.1 Connect Genius to Eclipse . . . 29
11.2 Insert example party . . . 30
11.3 Debugging . . . 30

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 Bids, Issues and Values

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.

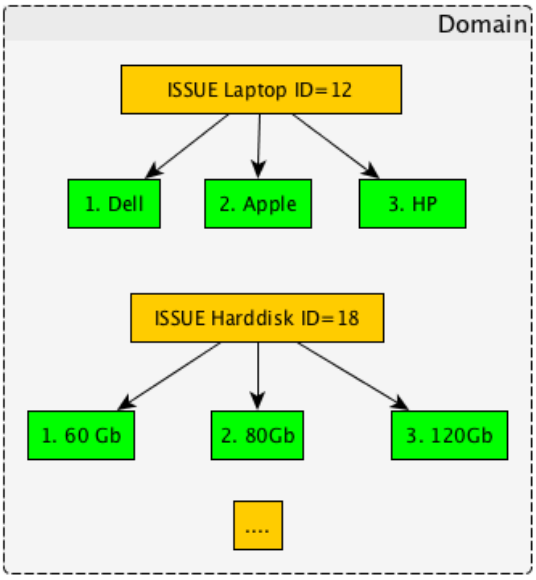

Figure 1: An example domain for laptop negotiation. Issues are orange, values are green

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

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 "60Gb", "80Gb" and "120Gb". These issues are all instance of *IssueDiscrete*. A valid bid in the laptop domain is e.g. `Laptop:Dell, Harddisk: 80Gb, monitor:17"`. A bid `Laptop Asus, Harddisk: 80Gb, monitor:17"` is not a valid bid because Asus is not a valid issue value in the example domain, and `Laptop Asus, Harddisk: 80Gb, CPU:i7"` is not valid because CPU is not an issue in this domain.

1.2 Preference Profile, Utility Space

The *Preference Profile* describes how bids are preferred over other bids. Typically, each participant in a negotiation has his own preference profile. Genius supports utility spaces and partially ordered profiles.

1.2.1 Utility spaces

One form of profile is the *UtilitySpace*. The UtilitySpace specifies the preferences using an *evaluator*: 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. So a bid is preferred if and only if it has a higher utility than another bid.

A common form of the Utility space is the *Linear Additive Utility Space*. This space is additive because each of the issue values in the domain have their own utility of their own, and all the sub-utilities just add up for the total utility of the bid. For instance, we like Apple with utility evaluation 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.

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

Other types of UtilitySpaces are the *ConstraintUtilitySpace* and the *NonlinearUtilitySpace*. These are more experimental and not described here in more detail.

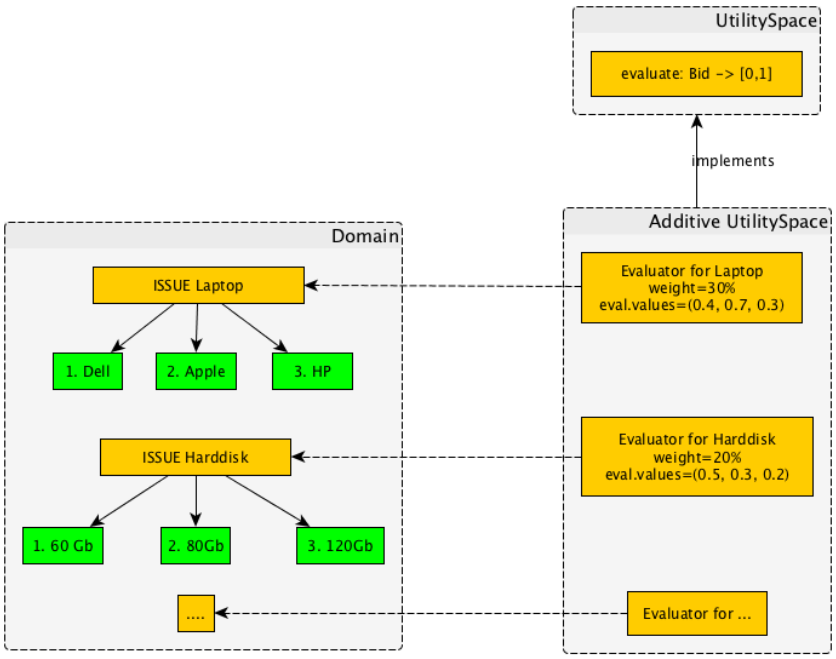

Figure 2: An example additive utility space for the laptop domain.

1.2.2 Partially ordered profile

The *UncertainAdditiveUtilitySpace* is a profile type uses partially ordering (instead of assigning a utility value to bids). In a partial ordering, the available information is that bid X is preferred over bid Y for a subset of the possible bids. For research purposes, this profile is generated from a AdditiveUtilitySpace as will be described in the section 4, but the underlying AdditiveUtilitySpace is normally not visible for the party that uses the profile. The generation of the partial ordering works as follows. The values *comparisons*, *errors* and *experimental* are additional parameters of the UncertainAdditiveUtilitySpace that control the generation.

1. a subset of *comparisons* bids are selected randomly from all possible bids.
2. the selected bids are sorted in ascending utility

Notice: AbstractNegotiatonParty on initialization will do a simplistic attempt to convert an UncertainAdditiveUtilitySpace into an AdditiveLinearUtilitySpace.

Warning: if the number of possible bids is large, iterating over all bids in the space and sorting them can run out of memory and crash GENIUS .

1.3 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 all parties in the negotiation. Identifying these special bids may lead to a better agreement for both parties.

For a single party, the optimal bid is the bid that is most preferred / has maximum utility. Often this bid is not preferred so much / has a low utility for other parties, and therefore the chance of agreement is low. A more general notion of optimality of a negotiation involves the utility of all parties.

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 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 parties can hide their preferences. This entails that an party 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 party 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.

1.4 Negotiation Protocol

The negotiation protocol determines the overall order of actions during a negotiation. Parties 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.

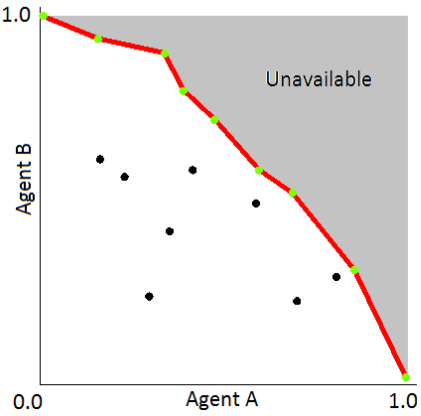

Figure 3: A point indicates the utility for both parties of a bid. The red line is the Pareto optimal frontier.

1.5 Reservation Value

A reservation value is a real-valued constant that sets a threshold below which a rational party 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 minimum acceptable utility, offers with a utility would normally not be accepted by a party. Reservation values typically differ for each negotiation party. In case no reservation value is set in a profile, it is assumed to be 0. Notice that if a negotiation ends with no agreement, parties normally get a utility of 0, regardless of the reservation value.

1.6 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 party 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 parties, the discount generally differs per party. 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 a party; 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:

$$U_D^t(\omega) = U(\omega) \cdot d^t \tag{2}$$

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 parties to reach an agreement. Note that discount factors are part of the preference profiles and therefore different parties 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 parties to end the negotiation early and thereby default to the reservation value.

Note: time pressure has little meaning if the profile is not defined in terms of utilities, eg a partially ordered profile.

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 a party 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 party'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 `genius.core.protocol` and have the names matching the subsections below.

2.1 Stacked Alternating Offers Protocol

According to this protocol [**?**] , 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 [**?**] , all parties have a bid from all parties 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 parties 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 party's 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 parties 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 parties. 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 [**?**].

This protocol uses the following turns in every round:

1. Mediator proposes an OfferForFeedback.
2. The other parties (not the mediator) place a GiveFeedback, indicateing 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 for the parts of the session outside the scope of the protocol specification.

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 party. Then the party's init code is called to set up the party. 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 parties 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 party 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.

3 Install and Run GENIUS

GENIUS can run on any machine running Java 8. Java 9 is not yet supported. 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 folder "genius-XXX" containing the following files:

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

You start GENIUS by double-clicking the genius-XXX.jar file, or using "open with" and then selecting Java.
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, parties, and BOA components.
- The **Status Window** shows the negotiation status or selected domain/preference profile.

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.

In some rare cases, parties 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`. But usually, if your party runs out of memory then there is some design flaw or bug. Competitions usually are run with the default amount of java memory so it is recommended to ensure that your party performs properly without requiring additional memory.

Please refer to chapter 9 for instructions on running GENIUS in debug mode to debug your party.

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

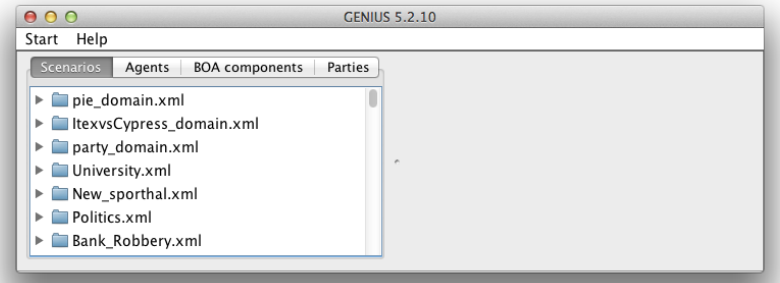

Figure 4: GENIUS right after start-up. The left half is the components panel, the right half the status panel.

4 Scenario Creation

GENIUS offers tools to create Domains and Profiles. Currently GENIUS supports editing the Additive and the UncertainAdditive utility space This section discusses how to create domains and preference profiles.

4.1 Creating a Domain

By right clicking on the list of available scenarios in the Domains panel a popup menu with the option to create a new domain is shown. After clicking this option a pop-up appears requesting the name for the new domain. After you enter a name and click ok, the new domain is 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.

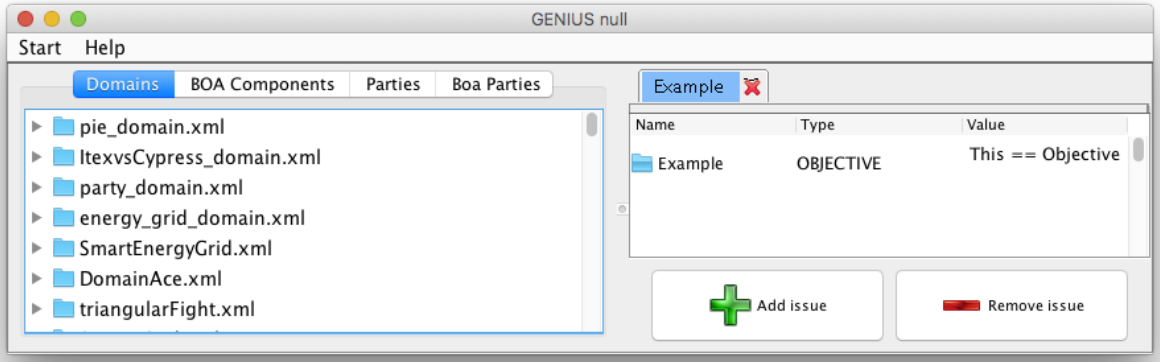

Figure 5: GENIUS after creating a new Example domain.

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 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.

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.

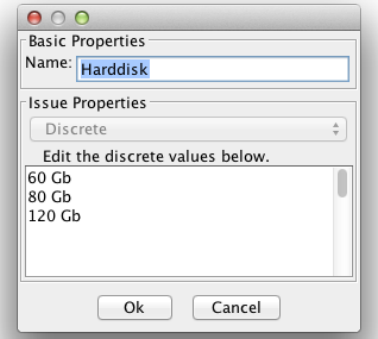

Figure 6: Creating a discrete issue.

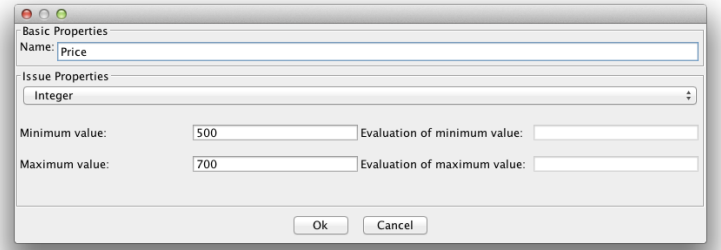

Figure 7: Creating an integer issue.

4.2 Creating an AdditiveUtilitySpace

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.

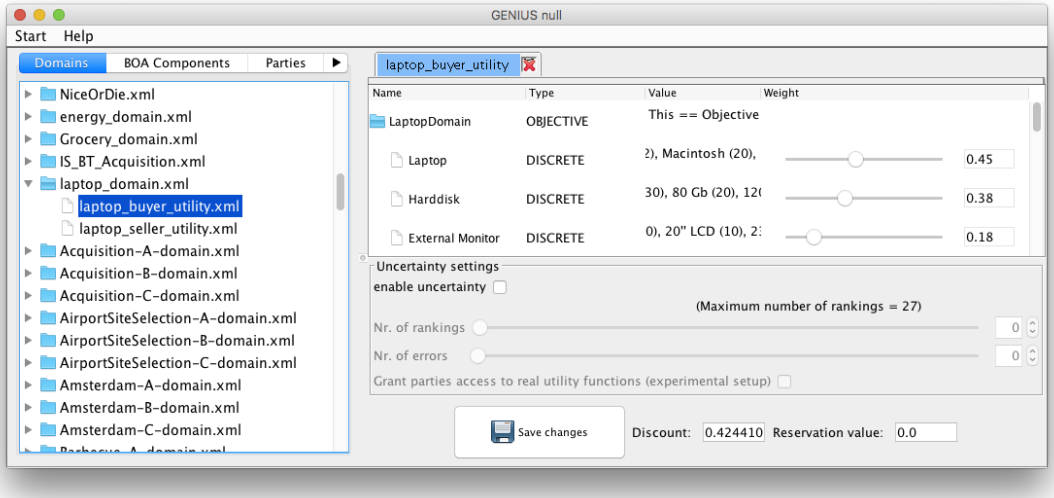

Figure 8: GENIUS after creating a new utility space.

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. Make sure that you leave the "enable uncertainty" checkbox unchecked.

1. 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.
2. 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 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.

3. The final step is to set the reservation value and discount of a preference profile.
4. If you are satisfied with the profile you can save it by pressing “Save changes”.

4.3 Creating an UncertainAdditiveUtilitySpace

To create an uncertain additive utility space, first make an AdditiveUtilitySpace (section 4.2). Check the ”Enable uncertainty” checkbox (figure 8).Set the nr of rankings and Nr. of errors as required. If your profile is for testing purposes, also check ”Grant parties access to real utility functions”. The details of these settings are explained in section 1.2.2. Press ”Save changes” to store your profile.

5 Running Negotiations

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

- **Session**. A single negotiation session in which a number of parties negotiate.
- **Tournament**. A tournament of multiparty sessions.

you start one of these by selecting them from the Start menu (Figure 4).

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

In contrast, non-automated parties are parties that are fully controlled by the user. These types of parties 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 Session

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

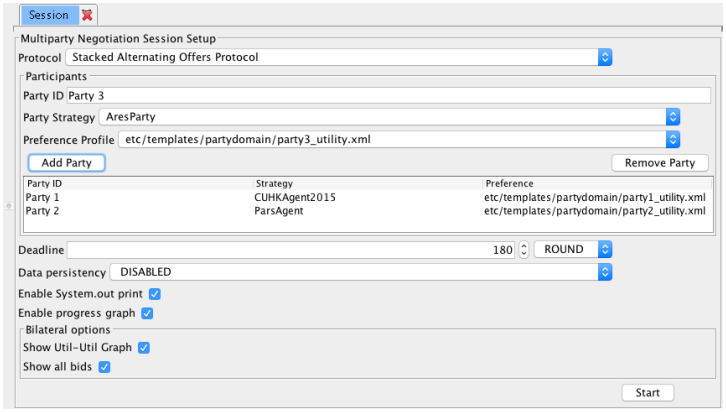

Figure 9: A multi-party negotiation session.

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 parties 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 **??**)

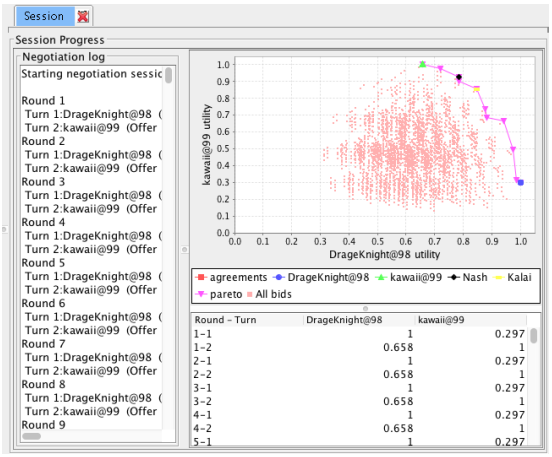

Figure 10: Bilateral progress panel.

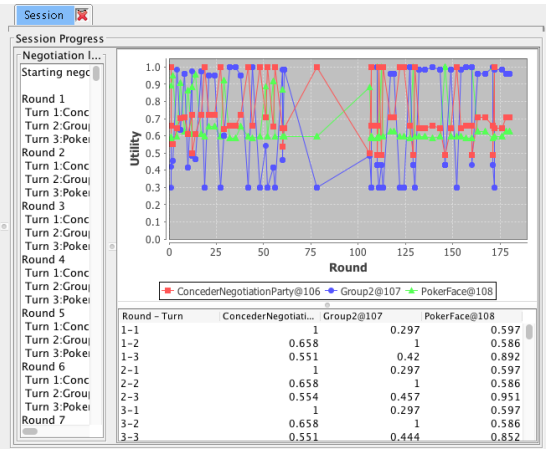

Figure 11: Multilateral progress.

5.2 Running a Tournament

A tournament is a set of sessions. To prepare a tournament, select "Start" and then "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
- **Data persistency**. The type of persistent data available to the parties. Same options as in section 5.1.

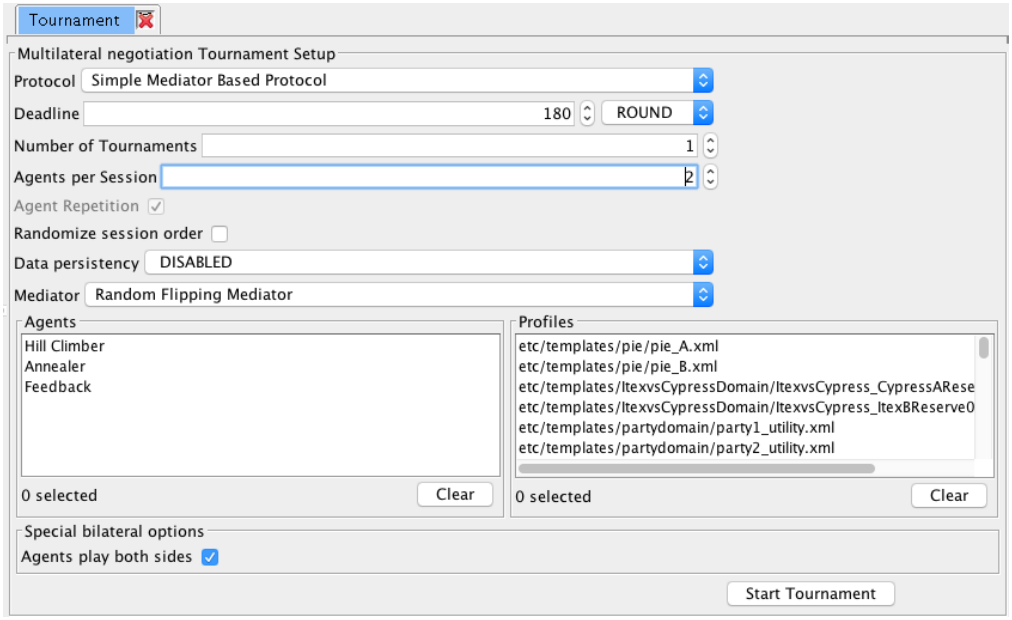

Figure 12: Tournament

- **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.

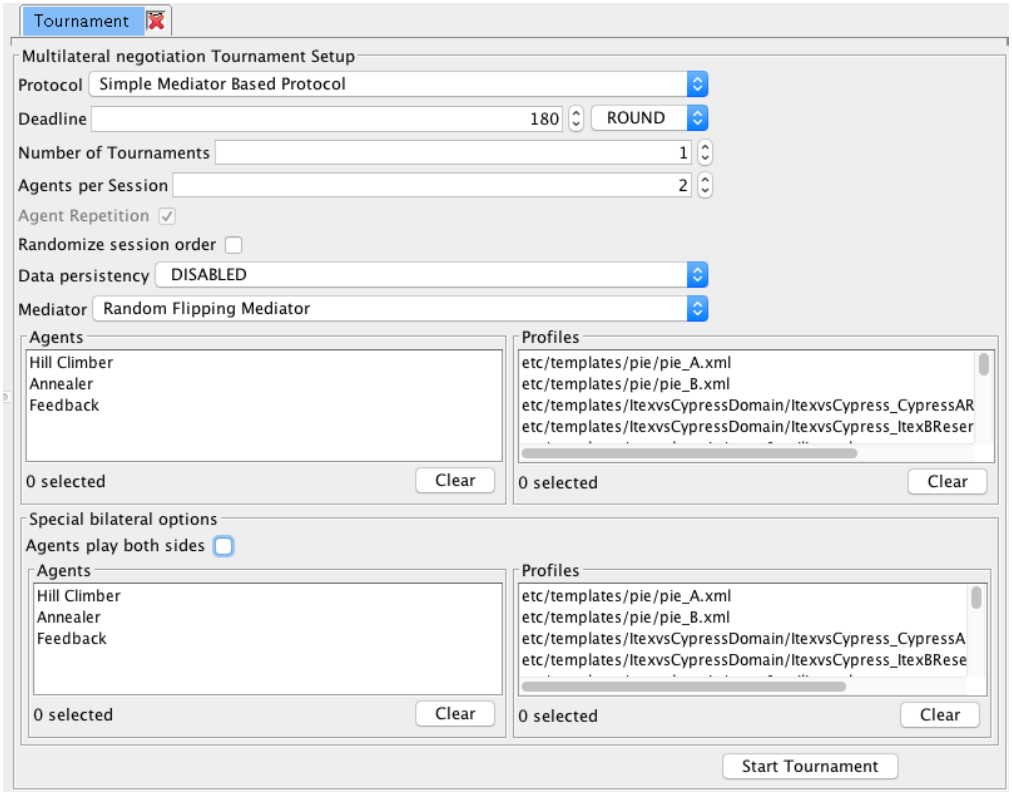

Figure 13: Bilateral Tournament

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.

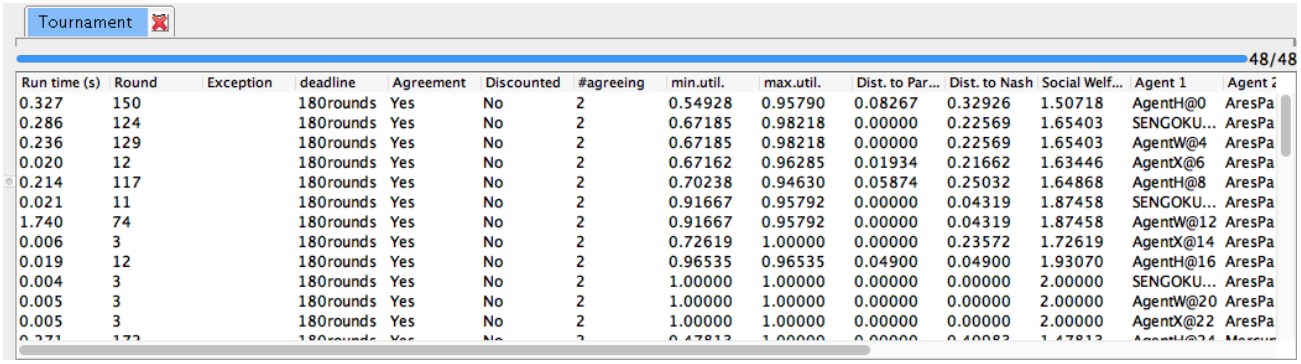

Figure 14: Tournament Progress panel

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 **??**)

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.
- **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 a party 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 parties to draw for each session. This excludes a possible mediator.
- **Party Repetition** true if parties are to be drawn from the parties 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 parties per session and the parties 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 party pool A, with our 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

Genius logs quality measures to log files. Logs are written both in `.csv` and `.xml` format. Logs are written to the `log/` directory. Filenames contain the date and time the session/tournament started.

The output of the log files differs, depending on whether you ran a tournament or a single session. The following subsections discuss the output for these.

Party names are printed as follows. If the party is a normal NegotiationParty, then the party will print out as something like `Atlas32016@2`. The part before the '@' is the party's name, the part after the '@' is added by the runner to make the name unique. If the party is a BOA party, the name is "boa-" followed by the concatenation of its offering strategy, acceptance condition, opponentmodel and omstrategy. For example you may get this as boa party name:

```
boa-genkus.core.boaframework.offeringstrategy.anac2012.CUHKAgent_Offering-
genius.core.boaframework.acceptanceconditions.anac2012.AC_CUHKAgent-
genius.core.boaframework.opponentmodel.CUHKFrequencyModelV2-
genius.core.boaframework.omstrategy.TheFawkes_OMS@3
```

6.1 Session logs

Both the XML and CSV log files from a session get the filename `Log-Session_` followed by day and time. The contents however differ.

6.1.1 Session CSV file

The `.csv` file contains one line for each turn, like this: `1,1,0.0055248618784530384,AgentHP2_main@0,(Offer  bid:Bid[Food: Chips`

The columns are, in order:

1. round number.
2. turn number
3. the time of agreement, in the range [0,1] where 0 means the start of the session and 1 the maximum time/number of rounds allowed for the negotiation.
4. the party that acted.
5. the action that the party did. The action consists of the action type name ("Offer", "Accept", "EndNegotiation", etc) and the bid details if available.

6.1.2 Session XML file

When running a session, the XML file contains only the details of the final outcome of the negotiation.

The fields in the `NegotiationOutcome` element:

1. currentTime: the moment when the final outcome was available.
2. timeOfAgreement, agreement time in the range [0,1] where 0 means the start of the session and 1 the maximum time/number of rounds allowed for the negotiation.
3. lastAction: the last action that was done
4. the domain that was being run
5. bids: the total number of bids that have been done in this session
6. the total run time in seconds
7. the outcome : the final accepted bid , or "-" if there was no final agreement
8. the startingAgent,
9. the deadline = maximum amount of time/rounds for this session.

and then, for each of the parties that participated a "resultsOfAgent" element containing:

1. the party's name.
2. the party's description
3. the party's utilityspace filename
4. the full party class path
5. the final utility (un-discounted) of this party, or 0 if there was no agreement
6. the final utility (discounted)
7. the value of discount(1,1). This is the remaining utility that a un-discounted utility of 1 would have at the end time (t=1). For default discount formulas, this equals the 'discount factor'.

Here's an example

```
<?xml version="1.0"?>
<Session>
<NegotiationOutcome currentTime="Thu Mar 08 12:50:33 CET 2018"
    timeOfAgreement="0.3425414364640884"
    lastAction="(Accept  bid:Bid[Food: Chips and Nuts, Drinks: Handmade ...oap, ])"
    domain="etc/templates/partydomain/party_domain.xml" bids="175" runtime="0.652346172"
    finalOutcome="Bid[Food: Chips and Nuts, Drinks: .....d Soap, ]"
    startingAgent="-" deadline="180rounds">
<resultsOfAgent agent="AgentHP2_main@0" agentDesc="ANAC2016 agenthp2"
    utilspace="etc/templates/partydomain/party1_utility.xml"
    agentClass="agents.anac.y2016.agenthp2.AgentHP2_main"
    finalUtility="0.8483333333333334" discountedUtility="0.8483333333333334" discount="1.0">
</resultsOfAgent>
<resultsOfAgent agent="AgentLight@1" agentDesc="ANAC2016 agentLight"
    utilspace="etc/templates/partydomain/party2_utility.xml"
    agentClass="agents.anac.y2016.agentlight.AgentLight"
    finalUtility="0.8564909885811369" discountedUtility="0.8564909885811369" discount="1.0">
</resultsOfAgent>
<resultsOfAgent agent="Atlas32016@2" agentDesc="ANAC2016 Atlas3"
    utilspace="etc/templates/partydomain/party3_utility.xml"
    agentClass="agents.anac.y2016.atlas3.Atlas32016"
    finalUtility="0.6924080946242358"
    discountedUtility="0.6924080946242358" discount="1.0">
</resultsOfAgent>
</NegotiationOutcome>
</Session>
```

6.2 Tournament logs

All log files from a tournament get the filename `tournament-` followed by day and time followed by the domain name and an extension. There are 3 log files created: a log.csv file, a log.xml file and a logStats.xml file.

If you terminate a tournament before it completes, the .log. files will be written up to the last completed session and there will be no logStats file.

6.2.1 Tournament log.csv file

tournament .csv files start with these a line containing `sep=;` indicating that we use the comma as separator character for fields. Then there is a table header typically looking like this (if there are 3 parties in each session, and all this on 1 line)

```
Run time (s);Round;Exception;deadline;Agreement;Discounted;#agreeing;min.util.;max.util.;
  Dist. to Pareto;Dist. to Nash;Social Welfare;
  Agent 1;Agent 2;Agent 3;
  Utility 1;Utility 2;Utility 3;
  Disc. Util. 1;Disc. Util. 2;Disc. Util. 3;
  Profile 1;Profile 2;Profile 3
```

The rest of the log file contains one line for each final session outcome, matching the columns in the header:

1. the run time of that session (seconds).
2. the number of rounds that were completed
3. the exception message, if an exception occured
4. the deadline = maximum amount of time/rounds for this session.
5. whether an agreement was reached (Yes) or not (No).
6. whether there was a discount factor (i.e. discount(1,1) is not 1) (Yes or No).
7. the final number of agreeing parties
8. the minimum utility achieved by the parties
9. the maximum utility achieved by the parties
10. the distance to the pareto curve (the nearest bidpoint on the pareto)
11. the distance to the nash optimum point
12. the distance to the social welfare point
13. the names of all parties
14. the un-discounted utilities of all parties
15. the discounted utilities of all parties

16. the profile names of all the parties

For example, one line of the output can look like this (all on 1 line)

```
4.965;173;;180rounds;Yes;No;3;0.58083;0.95256;0.00000;0.44991;2.13706;
ClockworkAgent@14;Farma@15;Caduceus@16;
0.5808333333333333;0.6036696609166442;0.9525594478616071;
0.5808333333333333;0.6036696609166442;0.9525594478616071;
party1_utility.xml;party2_utility.xml;party3_utility.xml
```

6.2.2 Tournament log.xml file

The .log.xml file contains one `<NegotiationOutcome>` element for each completed round. These elements are formatted exactly as in 6.1.2.

6.2.3 Tournament logStats.xml file

The logStats.xml file contains for each of the parties that participated in the tournament statistical info:

1. agentname: the party's name (full class path)
2. totalUndiscounted: the total sum of the un-discounted utilities that it achieved
3. totalDiscounted: the total sum of the discounted utilities that it achieved
4. numberOfSessions: the total number of sessions that it participated in
5. totalNashDist: the accumulated distances to the Nash Point
6. totalWelfare: the accumulated distances to the Social Welfare Point
7. totalParetoDistance: the accumulated distances to the Pareto frontier
8. meanDiscounted: totalDiscounted / numberOfSessions
9. meanUndiscounted: totalUndiscounted / numberOfSessions
10. meanNashDistance: totalNashDistance / numberOfSessions
11. meanWelfare: totalWelfare / numberOfSessions
12. meanPareto: totalPareto / numberOfSessions

7 Creating a Negotiation Party

To create an negotiation party, we suggest to follow the instructions in the Appendix and start with one of the examples. You can then proceed by changing the example.

7.1 Example Parties

The GENIUS zip file contains two examples: the Multiparty example and the Storage example. To compile an example, set up your workspace as in the appendix (Section 11) and copy an example folder into `src/`.

The multiparty example just accepts any acceptable bid with a random probability of 0.5.

The storage example demonstrates using 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.

To run this example, you need to set up GENIUS to allow persistent data storage (the default is off). In the GENIUS tournament setup panel, use 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 party gets a different profile every other run and thus there are persistent data stores, one for each profile.

Method	description
init	Initializes the party, informing it of many negotiation details. This is be called exactly once by the negotiation system, immediately after construction of the class
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 NegotiationParty chose an Action.
getDescription	Returns a human-readable description for this party
getProtocol	The actual supported MultilateralProtocol. Usually this returns StackedAlternatingOffersProtocol. Your party should override this if it supports a another protocol
negotiationEnded	This is called to inform the party that the negotiation has been ended. This allows the party to record some final conclusions about the run

Table 1: Methods of NegotiationParty. Check the javadoc for all the details

7.2 Implementing NegotiationParty

This section discusses details of implementing a NegotiationParty.

Every party must at least implement the `genius.core.parties.NegotiationParty` interface (Table 1), Also the 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 `genius.core.parties.AbstractNegotiationParty` which is a basic implementation of NegotiationParty. This class also provides convenient support functions for building your party.

Your party might need to check the exact type of the provided AbstractUtilitySpace (inside NegotiationInfo), for instance if your party supports for example only AdditiveutilitySpace. Also check the provided UserModel, if that is set (not null) then this overrides the value returned by getUtilitySpace..

A number of useful classes is given in 2. The javadoc contains the full details of all available classes. We recommend to use the javadoc included with the distribution to check the details of all the involved classes. Notice that some classes, e.g. SortedOutcomeSpace, may take a long time and large amounts of memory to sort a large bid space, which may exceed the available time and space for your party. Therefore these methods should be used with caution.

`NegotiationInfo` The context of the negotiation: the partial profile ("UserModel"), utility space, the timeline and deadline, the agentID and persistent data container.
`UtilitySpace` The preference profile of the scenario allocated to the party. It is recommended to use this class when implementing a model of the opponent's preference profile.

`Timeline`
Use timeline for every time-related by using `getTime()`.
`Action chooseAction(List<Class<? extends Action>> possibleActions)`
This function should return the action your party wants to make next.

`Action` Superclass of negotiation actions like Offer, Accept and EndNegotiation..
`BidHistory` a structure to keep track of the bids presented by the party and the opponent.
`SortedOutcomeSpace` a structure which stores all possible bids and their utilities by using BidIterator. In addition, it implements 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. WARNING (1) SortedOutcomeSpace iterates over all bids and thus might be unusable in large bidspaces (2) Some parties have created their own copy of SortedOutcomeSpace, so be careful to pick the genius.core version.
`BidIterator` a class used to enumerate all possible bids. Also refer to *SortedOutcomeSpace*.
`BidDetails` a structure to store a bid and its utility.

Table 2: Important classes used for creating a NegotiationParty.

7.2.1 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;
}
```

7.2.2 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 parties try to model the opponent's strategy or preference profile.

```
public Action chooseAction() {
    Action action = null;
    Bid partnerBid = null;
    try {
        if (actionOfPartner == null)
            action = chooseRandomBidAction();
        if (actionOfPartner instanceof Offer) {
            partnerBid = ((Offer) actionOfPartner).getBid();
            double offeredUtilFromOpponent = getUtility(partnerBid);
            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(), partnerBid);
        }
        if (timeline.getType().equals(Timeline.Type.Time)) {
            sleep(0.005); // just for fun
        }
    } catch (Exception e) {
         // best guess if things go wrong. Notice this may still fail
        action = new Accept(getAgentID(), partnerBid);
    }
    return action;
}
```

The method *isAcceptable* implements the probabilistic acceptance functionP_{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 15 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.

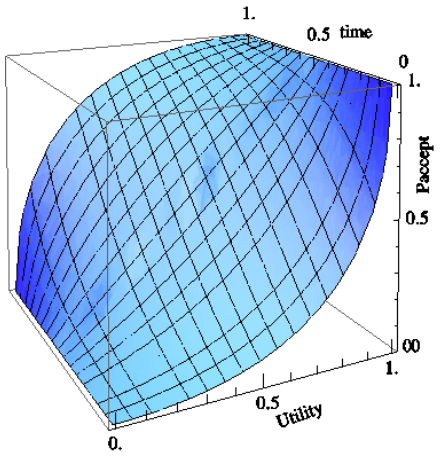

Figure 15: P_{accept} value as function of the utility and time (as a fraction of the total available time).

7.3 Implementing a party with uncertainty

In order to program an agent with uncertainty, we recommend that your agent extends the *AbstractNegotiationParty* class. That class has some support functions that load the normal utilityspace with an approximation that is useful for plotting outcomes.

7.3.1 Overriding functions

In order to change the way your agent handles the uncertainty, you can override the *estimateUtilitySpace()* function. This function in AbstractNegotiationParty returns an *AbstractUtilitySpace* object. In the *AbstractNegotiationParty* class, this code looks as follows:

```
public AbstractUtilitySpace estimateUtilitySpace() {
    Domain domain = getDomain();
    AdditiveUtilitySpaceFactory factory
        = new AdditiveUtilitySpaceFactory(domain);
    BidRanking bidRanking = userModel.getBidRanking();
    factory.estimateUsingBidRanks(bidRanking);
    return factory.getUtilitySpace();
}
```

As can be seen from the function, a custom *AbstractUtilitySpace* is created using the domain and the bid ranking. This function tries to approximate the utility function using a simple counting heuristic. This heuristic does not perform very well, so there is need to implement your own function. An example is included in the *UncertaintyAgentExample*. In this example, the *AbstractNegotiationParty* class is extended and the *estimateUtilitySpace()* function is overridden. Overriding this function can be done as follows:

```
@Override
public AbstractUtilitySpace estimateUtilitySpace()
{
    return new AdditiveUtilitySpaceFactory(
                getDomain()).getUtilitySpace();
}
```

This function overrides the standard function and implements its own method to estimate the Utility space. Currently, this returns a utility function with equal weights and values set to zero. To estimate the utility function, you can use the *BidRanking* class. The bid ranking for the current session can be accessed using *userModel.getBidRanking()*. The next section will show what information is included in the *BidRanking*.

7.3.2 Bid Ranking

When running an uncertain agent, all utility information is given through a bid ranking. This bid ranking consists of an ordering of different bids for the current domain. The ranking that the agent receives is ordered from low utility to high utility. This ranking can be used to estimate a utility function.

The *BidRanking* class consists of a list of `Bids` To obtain the *Bid* classes from the *Bidranking*, you can use the *getBidOrder()* function on the *Bidranking* object. This object is obtained in the agent using *userModel.getBidRanking()*. Check the javadoc for more details, and we suggest you check the source code of BidRanking.

To access the list of *Bid* objects directly, you can use the following snippet:

```
List<Bid> bids = userModel.getBidRanking().getBidOrder();
```

Note: The bids that are obtained from the *BidRanking* using the *getBidOrder()* function are listed from low utility to high utility. This means that the first element in the list has the lowest utility score. When iterating over the list, every next bid will be either valued **higher or the same** as the current bid in the list.

7.3.3 Accessing the real utility space for debugging

When the Experimental Setup checkbox is enabled (Figure 5), your agent can access the real utility function. This can be used to verify the utility function that you have created. In order to get access to this function, the userModel should be cast to an *ExperimentalPairwiseComparisonUserModel* object. This can be done as follows:

```
ExperimentalPairwiseComparisonUserModel e =
    (ExperimentalPairwiseComparisonUserModel) userModel;
AbstractUtilitySpace realUSpace = e.getRealUSpace();
```

Now, you will have access to the real utility space with *realUSpace*.

7.3.4 Uncertainty agent checklist

This section will give a short overview of what to do in order to enable your agent to work with uncertainty. You should take the following steps:

1. Extend the *AbstractNegotiationParty* class.
2. Override the *estimateUtilitySpace()* function that returns an *AbstractUtilitySpace* class.
3. Using the *getDomain()* function and the *BidRanking*, create an estimation for the utility function. (E.g. Counting, Machine Learning, Statistical methods, etc.)
4. Implement the normal methods necessary for the agent to do the bidding. This is the same as for normal agents, the uncertainty is only used on startup of the agent in order to estimate the utility function.

7.4 Loading a NegotiationParty

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

Locate the Parties repository tab in the GUI (Figure 16). Right click in this area and select "Add Party". A file browser panel pops up. Browse to your compiled `.class` file that implements the NegotiationParty and select it. Typically Eclipse compiles into `bin`. Your party will appear at the bottom of the parties repository. The `partyrepository.xml` file is automatically updated accordingly.

To do this manually without using the GUI, quit GENIUS , open the `partyrepository.xml` file [2] and add a section like this

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

After that you can restart GENIUS so that it loads the new party.

[2] This file is automatically created the first time you run GENIUS

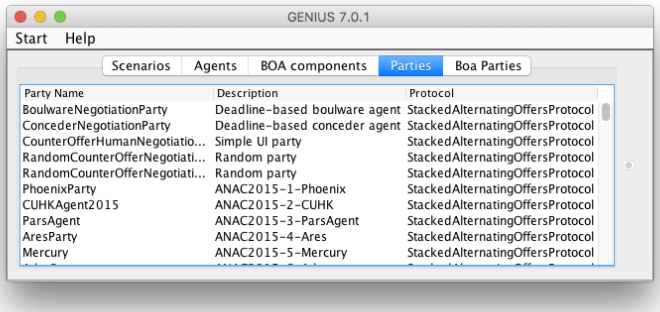

Figure 16: The parties repository.

7.5 Third party code

You should not use maven or jars to add dependencies for your party. The reason is GENIUS or other parties might already have another version of your library in use. Java 8 can not deal properly with multiple versions of the same library in within a JVM. The result would be inconsistent, incorrect or buggy behaviour, or even crashes.

Instead, 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 party, 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 party on the specific version of the library that your party needs and to avoid version conflicts (java will run an unspecified version of the library in case of conficts).

8 Creating a BOA Party

Instead of implementing your negotiating party from scratch, you can create a BOA party using the *BOA framework*. The BOA negotiation party architecture allows to reuse existing components from other BOA parties. Many of the sophisticated party 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 (*offering 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 a party 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*.

WARNING
Many of the provided BOA components currently assume a single opponent party. These will behave incorrectly when used with multiple opponents.
We recommend checking the source code of the BOA components you want to use, or write your own components if you are creating a NegotiationParty.

8.1 Components of the BOA Framework

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

Offering strategy An offering strategy is a mapping which maps a negotiation trace to a bid. The offering 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 17). When receiving a bid, the BOA party 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 *offering strategy* determines the counter offer by first generating a set of bids with a similar preference for the party. The *offering strategy* uses the *opponent model strategy* to select a bid from this set taking the opponent's utility into account.

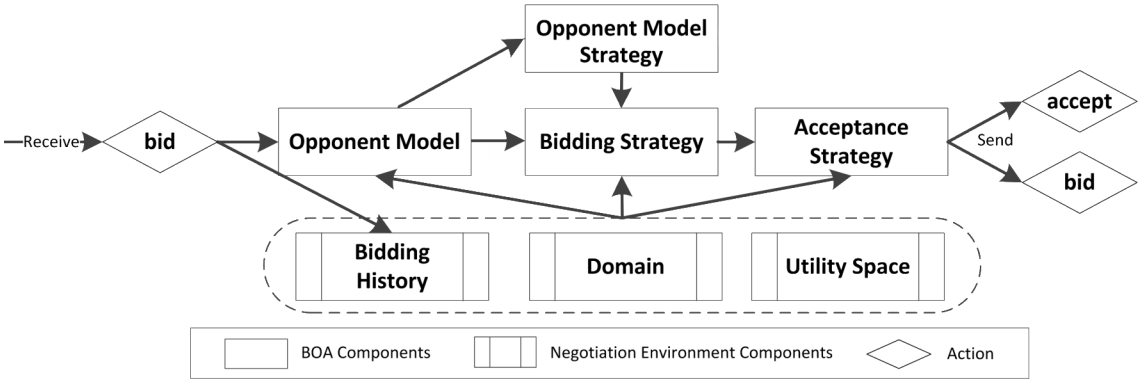

Figure 17: The BOA Framework Architecture.

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.

8.2 Create a BOA Party

A boa parties can be edited in the "Boa Parties" repository tab (Figure 18). Right-click in the panel to add items. Select an item and right-click to remove or edit an item.

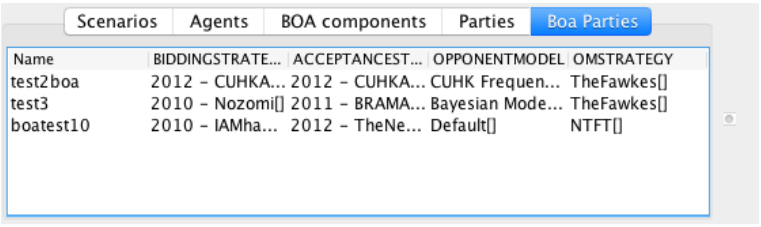

Figure 18: The BOA Parties repository tab.

After you selected to add or edit a BOA party (Figure 19). Here you can select a different Offering 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.

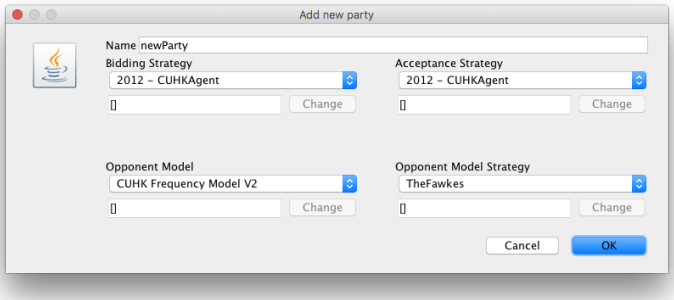

Figure 19: Editing a BOA party.

If you click on the "Change" button, another panel pops up where you can edit the parameters (Figure 20). 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 19). Then, parties 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 parties, 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 boa parties this way.

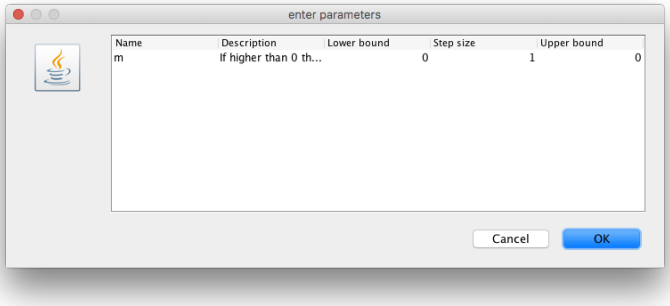

Figure 20: Editing the Parameters of a BOA party.

8.3 Creating BOA 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.

8.3.1 Set up a Workspace

BOA components must be compiled before they can be loaded into Genius. To compile a BOA component, follow the steps in (Section 11). Then, create your BOA code into `src`. For a quick start, you can copy the boaexample folder into `src`. Eclipse automatically compiles your BOA components into `bin`.

Please refer to chapter 9 for instructions on running GENIUS in debug mode to debug your components.

8.3.2 Add component to Genius

After your component has been compiled, you need to tell Genius where to find it. Go to the "BOA Components" tab and right click in the table. Select "Add new component". Enter the component name and click "Open". Browse to your compiled component and click "Open". Click ""Add component". After this, your component appears in the list and is ready for use.

8.3.3 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 3: 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.

8.3.4 Creating an Offering Strategy

An offering strategy can be easily created by extending the *OfferingStrategy* class. Table 4 depicts the methods which need to be overridden. The *init* method of the offering 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 partiess, 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 party can retrieve the value of the parameter by calling *parameters.get("b")*.

An approach often taken by many offering 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 party 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 offering strategy component.

8.3.5 Creating an Acceptance Condition

This section discusses how to create an acceptance strategy class by extending the abstract class *AcceptanceStrategy*. Table 5 depicts the two methods which need to specified.

`void init(NegotiationSession negotiationSession, OfferingStrategy offeringStrategy, OpponentModel opponentModel, Map<String, Double> parameters)` Method directly called after creating the party which should be used to initialize the component.
`Actions determineAcceptability()` Method which determines if the party 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.

8.3.6 Creating an Opponent Model

This section discusses how to create an opponent model by extending the abstract class *OpponentModel*. Table 6 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, Map<String, Double> parameters)` Method directly called after creating the party 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.

8.3.7 Creating an Opponent Model Strategy

This section discusses how to create an opponent model strategy by extending the abstract class *OMStrategy*. Table 7 provides an overview of the main methods which need to specified.

`void init(NegotiationSession negotiationSession, OpponentModel model, Map<String, Double> parameters)` Method directly called after creating the party 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.

8.4 Advanced: Converting a BOA Party to a Party

To convert a BOA party to a normal party you have to create a class that extends *BoaParty* and override the *init* method. Below is an example of a BOA party wrapped as a normal party. It's a bit hack-y because the BoaParty constructor

assumes all components known while an party often can decide this only at init time.

```
public class SimpleBoaParty extends BoaParty {

    public SimpleBoaParty() {
        super(null, new HashMap<String, Double>(), null,
                new HashMap<String, Double>(), null,
                new HashMap<String, Double>(), null,
                new HashMap<String, Double>());
    }

    @Override
    public void init(NegotiationInfo info) {
        SessionData sessionData = null;
        if (info.getPersistentData()
                .getPersistentDataType() == PersistentDataType.SERIALIZABLE) {
            sessionData = (SessionData) info.getPersistentData().get();
        }
        if (sessionData == null) {
            sessionData = new SessionData();
        }

        negotiationSession = new NegotiationSession(sessionData,
                info.getUtilitySpace(), info.getTimeline());
        opponentModel = new MyrequencyModel();
        opponentModel.init(negotiationSession, new HashMap<String, Double>());
        omStrategy = new NullStrategy(negotiationSession);
        offeringStrategy = new MyBiddingStrategy(negotiationSession,
                opponentModel, omStrategy);

        acceptConditions = new AC_Next(negotiationSession, offeringStrategy, 1,
                0);
        // we have init'd all params here, don't call super init
    }

    @Override
    public String getDescription() {
        return "Simple BOA Party";
    }
}
```

8.5 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 parties* 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 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));
        }
    }
}
```

9 Debugging

This section explains how to debug your party using Eclipse. It is assumed you set up your party already as in Chapter 7.

To run or debug your party as it runs in Genius, right click on the project root and select Run As.../Java Application. Then select `Application - genius` and click ok.

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

9.1 Source code and javadocs

The genius core source codes and javadocs are included in the genius.jar file. But if you like you can browse and download all sources at `https://tracinsy.ewi.tudelft.nl/pubtrac/Genius`.

10 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 [**?**].

11 Appendix

This appendix describes how to set up Eclipse to create and debug your own party.

11.1 Connect Genius to Eclipse

We expect that you installed Eclipse (Neon or higher) and Java version 8 on your computer.

1. Open the Eclipse Navigator with the menu Window/Show View/Navigator. You can close the Package Explorer.
2. Right click in the Navigator area and select New/Java Project. Create a new Java project. We name it `Group3assignment` but you can use any convenient name. Make sure you select "JavaSE-1.8" or equivalent to ensure your code will be java 8 compatible (Figure 21). Click Finish.

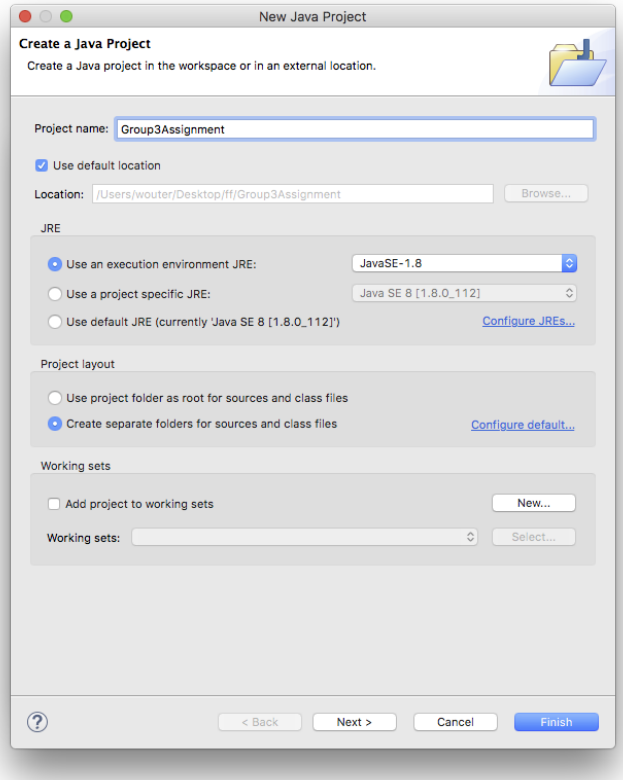

Figure 21: Create a new java project with the proper name and settings.

3. Drag the genius.jar file (from your unzipped download) into the project in the Eclipse Navigator area. Select "Copy files" and press OK.
4. Connect `genius` Jar:
   (a) Right click on the `Group3assignment` icon and select "Properties".
   (b) Select the Java Build Path.
   (c) Select the Libraries Tab.
   (d) Select "Add JARs", in the JAR Selection window (Figure 22).
   (e) Open the `Group3assignment` folder and scroll down to select `genius.jar`.
   (f) click a few times ok to close all dialog boxes.
5. Now you can run GENIUS as a Java application, by launching it as a `Application` (Figure 23). To do this, right click on the project, select `Run As`, select `Java Application` and then in the browser select `Application - genius`.

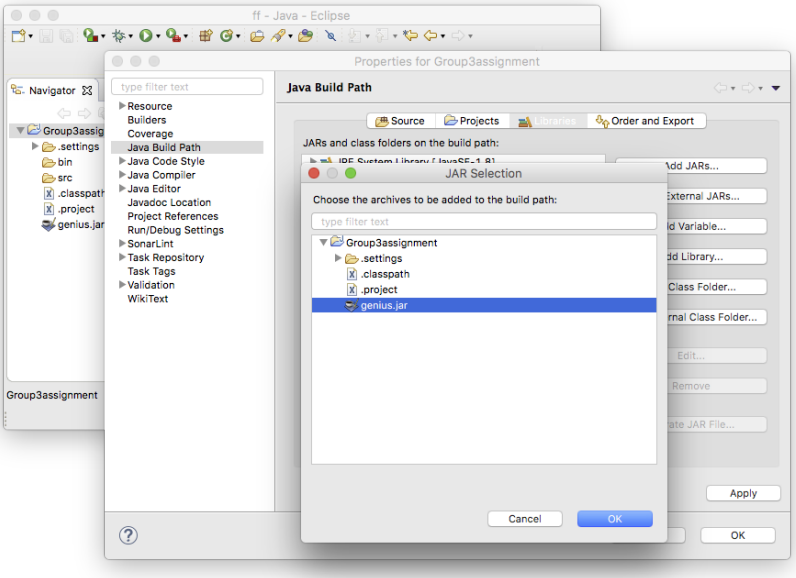

Figure 22: Attach the negosimulator jar to the project.

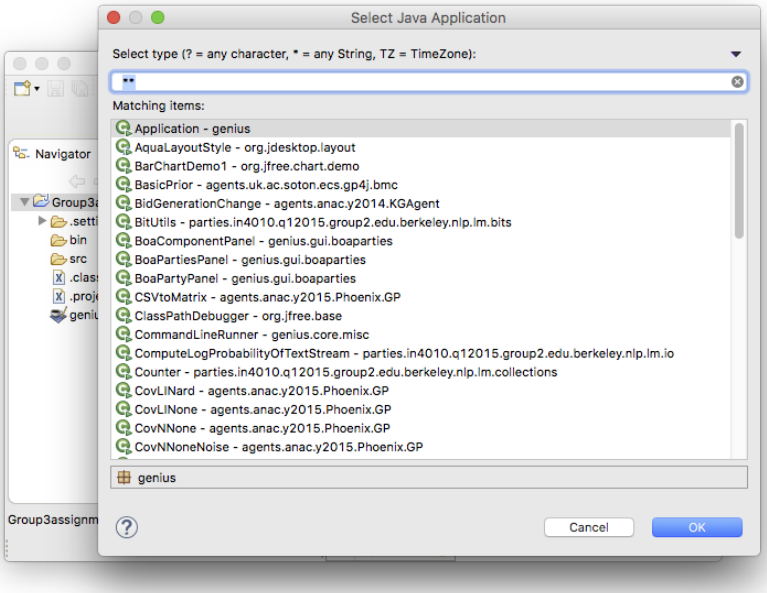

Figure 23: Starting Genius in Eclipse.

11.2 Insert example party

To compile an example party, just drag an example folder, eg storageexample, from your unzipped genius download entirely into the src folder in Eclipse. Select "Copy files and folders" and click ok.

11.3 Debugging

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