Negotiation User Guide - ANAC Version

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

January 12, 2012

In Genius, you can implement and analyze an agent that negotiates on your behalf. This document describes how you can install the required environment, work with the provided agents, and write, compile, and run such an agent yourself.

Contents

1 Theory Crash Course **1**
- 1.1 Actions . . . 1
- 1.2 Negotiation Protocol . . . 1
- 1.3 Reservation Value . . . 2
- 1.4 Time available for a Session . . . 2
  - 1.4.1 Discounting . . . 2
- 1.5 Negotiation Objects . . . 3
- 1.6 Utility of a Bid . . . 3
- 1.7 Optimality of a Bid . . . 4

2 Installing the Environment **4**
- 2.1 System Requirements . . . 4
- 2.2 Installation . . . 4
- 2.3 Progress & Error Messages . . . 5
- 2.4 Bug reporting . . . 5

3 Profile Creation **5**
- 3.1 Start the Negosimulator . . . 5
- 3.2 Load the Domain . . . 5
- 3.3 Create a Preference Profile . . . 6
- 3.4 Set the Evaluation Values . . . 7
- 3.5 Set the Issue Weights . . . 7
- 3.6 Save your Utility Space . . . 8

4 Running Negotiations **8**
- 4.1 Using an Automatic Agent . . . 8
- 4.2 Negotiation Results . . . 9
  - 4.2.1 The Utility Plot . . . 9
  - 4.2.2 Exchanged Offers . . . 9
- 4.3 Running a Tournament . . . 9

5 Writing a Negotiation Agent **9**
- 5.1 ReceiveMessage . . . 10
- 5.2 ChooseAction . . . 11
- 5.3 Compiling an Agent . . . 13

6 Data structures **13**

7 Conclusions **13**

8 References **13**

1 Theory Crash Course

This section gives a crash course on some essential theory needed to understand the negotiation system. The negotiation objects that are used in a negotiation and utility values are discussed.

1.1 Actions

In negotiation, the two parties take turns in doing the next negotiation action. The possible actions are:

ACCEPT	This action indicates that agent accepts the opponent's last bid.
OFFER	This action indicates that the agent proposes a new bid.
ENDNEGOTIATION	This action indicates that the agent terminates the entire negotiation, resulting in the reservation value being awarded to both players (see below; normally this is equal to the lowest possible score of zero for both agents).

1.2 Negotiation Protocol

The negotiation protocol determines the overall order of actions during a negotiation. Agents are obliged to stick to this protocol, and deviations from the protocol are caught and penalized. This section discusses the details of the protocol for this assignment.

Agent A and B take turns in the negotiation. One of the two agents is picked at random to start. When it is the turn of agent X (X being A or B), that agent is informed about the action taken by the opponent. If the action was an OFFER, agent X is subsequently asked to determine its next action and the turn taking goes to the next round. If it is not an OFFER, the negotiation has finished. The turn taking stops and the final score (utility of the last bid) is determined for each of the agents, as follows:

- the action of agent X is an ACCEPT. This action is possible only if the opponent actually did a bid. The last bid of the opponent is taken, and the utility of that bid is determined in the utility spaces of agent A and B. The opponent is informed of this accept via the ReceiveMessage method (but now without the subsequent chooseAction).
- the action returned is an ENDNEGOTIATION. The score of both agents is set to their reservation value.

So far for the protocol. If an agent does not follow this protocol, for instance by sending another action that is not one of the above or by crashing, the agents will also get their reservation values. If the agent deviates grossly from the protocol in such a way that it endangers the negotiation simulator itself, it may be disqualified entirely.

1.3 Reservation Value

A reservation value is a real-valued value that sets a threshold below which a rational agent should not accept any offers. Intuitively, a reservation value is the utility associated with a Best Alternative to No 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 by breaking off the negotiation (a non-zero reservation value makes breaking off potentially interesting) or by not reaching an agreement before the deadline associated with a negotiation session. 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, and both agents receive their reservation value instead.

Reservation values typically are different for each of the negotiating agents that negotiate in a session. In case no reservation value is set in a profile, the reservation value of an agent is set to zero (0). Given that utilities for outcomes range from 0 to 1, this essentially is the same as having no reservation value.

1.4 Time available for a Session

Normally, each negotiation session is allowed to last at most 180 seconds. If no agreement has been reached before this time, the negotiation will be terminated by killing the negotiation agents, and the utility of both parties will be their reservation value (normally 0). Only if (at least) one of the agents is a GUI agent requiring user input, the deadline is set to 1800 seconds. Notice that manipulation of the available time (by delaying the response of the agent) can be an important factor influencing the negotiation results, and one improvement for the example agent (SimpleAgent) would be to be more careful about this.

Note that in the negotiation environment, 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. Agents should not rely on the assumption that the negotiation time window corresponds to 180 seconds in real–time, but are required only to rely on the normalized time.

1.4.1 Discounting

Apart from a deadline, a scenario may also have *discount factors*. Discount factors devaluate the value of the issues under negotiation as time passes. Time is shared between the agents, but the discount is private to, and can different between each agent. The implementation of discount factors is as follows. Let d in $[0, 1]$ be the discount factor that is specified in the preference profile of an agent. Let t in $[0, 1]$ be the current normalised time, as defined by the timeline. We compute the discounted utility U_D^t of an outcome ω from the undiscounted utility function U as follows:

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

If $d = 1$, the utility is not affected by time, and such a scenario is considered to be undiscounted, while if d is very small there is high pressure on the agents to reach an agreement. Note that discount factors are part of the preference profiles and need not be *symmetric* (i.e. d can have different values for each agent).

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 reserve price, it may be rational for an agent to break off the negotiation early and take the outside option.

Note that, as opposed to having a fixed number of rounds, both the discount factor and deadline are measured in *real time*. This, in turn, introduces another factor of uncertainty since it is unknown to the agents how many negotiation rounds there will be, and how much time an opponent requires to compute a counter offer. Also, this computational time will typically change depending on the size of the outcome space.

1.5 Negotiation Objects

The central data structures in the negotiation are the bid and the utilityspace. Both work in a domain. Figure 1 shows an overview of the Domain and utility space data structures and their relations.

The **domain** describes which issues are the subject of the negotiation. To give a concrete example of a domain, in the laptop domain the domain is a list of issues, where the issues are laptop, harddisk and monitor. Issues are being referred to via the **issue ID**, a unique number for each issue. The domain description also describes the possible values that all the issues can take. In the laptop domain, all issues can have only discrete values, e.g. the harddisk issue can have only the values 60 Gb, 80 Gb and 120 Gb. These issues are an instance of IssueDiscrete. There are other types of Issue but discussion of them falls out of the scope of this short discussion.

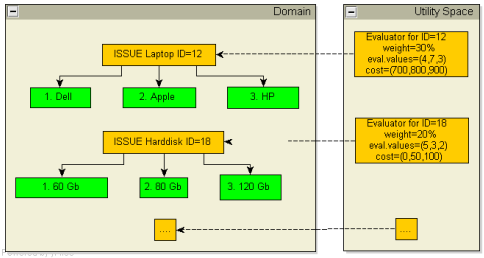

Figure 1: Overview of the data structures and relations.

Issues are an instantiation of a more general **Objective** class. The objective class itself is not relevant except that some functions return an Objective, and the returned object then has to be cast to an Issue or IssueDiscrete as needed.

The **utilityspace** provides all the information enabling the computation of the utility of some bid. It is implemented as a list of evaluators, one evaluator for every issue in the domain. Because the issues in the laptop domain are all discrete issues, the evaluators are all EvaluatorDiscrete objects.

The **Evaluator** object contains a weight, and for each value that the issue can have it gives an evaluation-value and a cost value.

The **weight** indicates the relative importance of an issue. The sum of the weights of all issues is 1.0.

The **evaluation-value** gives the evaluation, or utility, associated with each value that an issue can take. For instance, for the harddisk issue above, the evaluation values could be 2 for the 60 Gb, 4 for the 80 Gb and 10 for the 12 0Gb, etc.

The exact formula for computation of the utilities is given in Equation 3.

The **bid** is a set of values for each of the issues in the domain.

1.6 Utility of a Bid

A bid is a set of chosen values $v_1, \ldots, v_n$ for each of the N issues. Each of these values has been assigned an evaluation value $\text{eval}(v_i)$ in the utility space. Sometimes, there are also fixed costs $\text{cost}(v_i)$ associated with each value. The utility is the weighted sum of the normalized evaluation values, under the assumption that the cost is below the maximum cost of M. If the maximum cost is exceeded, the utility is zero.

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

And when costs are associated with the domain:

$$\text{Utility}(v_1, \ldots, v_n) = \begin{cases} U(v_1, \ldots, v_n) & \text{if CostSum}(v_1, \ldots, v_n) \leq M \\ 0 & \text{if CostSum}(v_1, \ldots, v_n) > M \end{cases} \tag{3}$$

$$\text{CostSum}(v_1, \ldots, v_n) = \sum_{i=1}^{N} \text{cost}(v_i) \tag{4}$$

1.7 Optimality of a Bid

For a single agent, the optimal bid is of course one with a maximum utility. But a more general notion of optimality of a negotiation involves the utility of both agents. Figure 2 shows the utilities of all bids for the two parties in the negotiation.

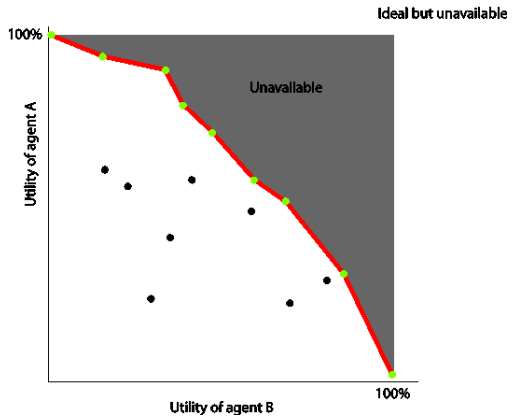

Figure 2: Utility plot. Each point indicates the utility for both agents of a particular bid. The red line is the Pareto optimal frontier.

There are multiple ways to define a more global "optimum". One approach to optimality is that a bid is not optimal for both parties if there is another bid that has the higher utility for one party, and at least equal utility for the other party. Thus, only bids in Figure 2 for which there is no other bid at the top right is optimal. This type of optimality is called Pareto optimality. The collection of Pareto optimal points is called the Pareto optimal frontier. Another approach is the Nash optimality. A Nash solution is a bid for which the product of the utilities of both agents is maximal.

2 Installing the Environment

2.1 System Requirements

You can run the negotiation environment on most systems running Java version 5 or 6. There are known issues with Linux (in particular Ubuntu and SUSE). You can download Java from the internet (www.sun.com/java) [sun07].

2.2 Installation

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

- `assignment.pdf`, containing the assignment;
- `userguide.pdf`, this document;
- `negosimulator.jar`, the negotiation simulator;
- a `templates` folder, containing various domain spaces, all with a few sample utility spaces (xml files);
- The `SimpleAgent.java` and `SimpleAgent.class` file.

2.3 Progress & Error Messages

When you run the negosimulator (by double-clicking the application), progress messages and error messages are printed mainly to the standard output. On Mac OSX you can view these messages by opening the console window (double-click on Systemdisk/Applications/Utilities/Console.app). On Windows this is not directly possible. Console output can be read only if you start the application from the console window

by hand, as follows. Go to the directory with the negosimulator and enter `java -jar negosimulator.jar` This will start the simulator, and all system.out messages will appear in the console window. You may see some errors and warnings that are non-critical.

2.4 Bug reporting

The negotiation environment has been tested extensively on Mac OSX 10.9 and on Windows XP, Windows 7. It should run on any machine running Java 1.5.0 or higher. This includes Solaris, Linux, and more recent versions of Mac OSX and Microsoft Windows. There are known issues with Linux (in particular Ubuntu and SUSE). There are still a number of known bugs in the negotiation environment, and possibly new bugs will be discovered during the course. Please be patient when you discover a bug and first try to find an alternative way first. If this is unsuccessful, please report the bug to `ai@mmi.tudelft.nl` or `W.Pasman@tudelft.nl`.

3 Profile Creation

The profile describes your personal preferences for a negotiation in a given domain. The profile is used to convert any bid in that domain to a value indicating how you would rate that bid. This is also called your utility of that bid. A profile is also called a utility space. Your utility space has a major impact on your final rating in the course, because it will be used to judge how well you completed the negotiations that you will perform. This section discusses how to edit your utility space.

3.1 Start the Negosimulator

Start the negosimulator by double-clicking the `negosimulator.jar` file in the negotiator package. The negosimulator screen is displayed in Figure 3. The Domains and Preference Profiles tab contains a repository of domains and preference profiles (under the corresponding domain). Domains and preference profiles can be added or deleted with the green + and red – buttons. The agents tab contains a repository of agents. Agents can also be added or deleted with the green + and red – buttons.

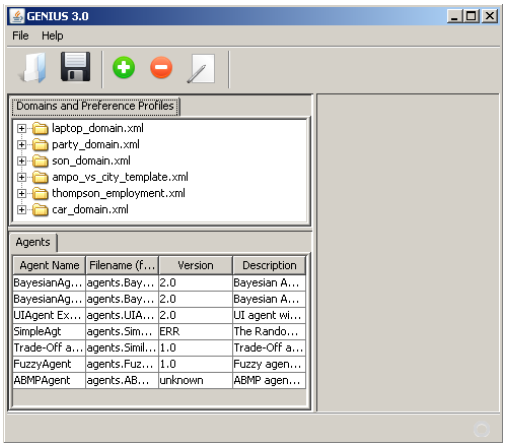

Figure 3: The negosimulator right after start-up.

3.2 Load the Domain

Load the domain specification that you want to work with, by double-clicking one of the domains in the Domains and Preference Profiles tab, for instance laptop_domain.xml. After loading the laptop domain the simulator will look like Figure 4. The left column shows the laptop domain, with the issues “Laptop”,

"Harddisk", and "External Monitor". The center column shows the type of the issue. The right column shows the values that are available for the issue.

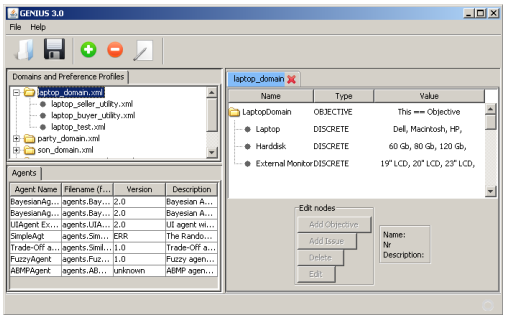

Figure 4: The negosimulator after loading the laptop domain.

3.3 Create a Preference Profile

We need to extend the domain that has now been loaded into an empty utility space for that domain. In the Domains and Preference Profiles tab, select the domain for which you want to create a new preference profile. Then select File > New > Preferences Profile. After this, the editor should look like Figure 5. New as compared to Figure 4 is the "Weight" column, showing the importance associated with this issue. Notice the numbers and check boxes at the far right. If you do not see them in your window, you may need to resize the window and the headers.

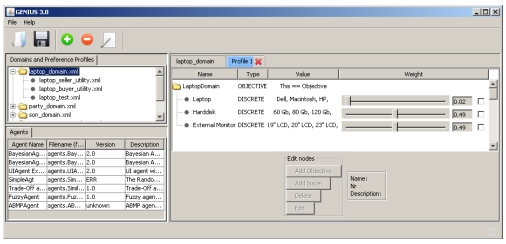

Figure 5: The negosimulator after creating a new utility space.

Now you are ready to start customizing your utility space to reflect your personal preferences. The next two steps will deal with this.

3.4 Set the Evaluation Values

In this step the utility values for each value will be set for each of the issues. Select one of the issues by clicking on the name of the issue (do not select the "LaptopDomain" which is not an issue). The selected name will turn blue and a large yellow area will appear. Then click the "Edit" button. The editor for discrete issues will pop up (see Figure 6).

Now you can edit the column Evaluation values by clicking in the blank column under "Evaluation values". You can enter the evaluation values, one per line, matching the values on your hand-written profile. Only integer numbers larger than 0 are allowed. To introduce a strong preference of one issue over other issues, just make the number very large. The window will look like Figure 7 after editing.

Do not edit the first column. Doing so will change the domain, resulting in a utility space that does not fit the laptop domain. The other columns should not be edited either for the same reasons. The editor will not allow editing of these values anyway, but you can also edit the xml files by hand in which case you have

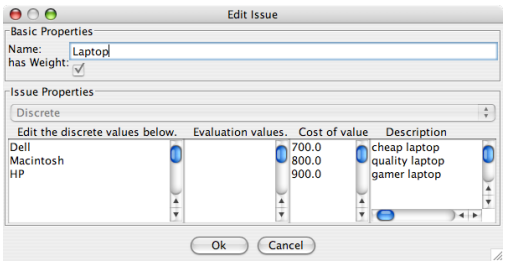

Figure 6: The discrete-issue editor.

to be careful not to touch those fields. When adjusting the evaluation values, keep in mind that the utility of a bid will be zero if the cost constraint is violated (see Equation 1).

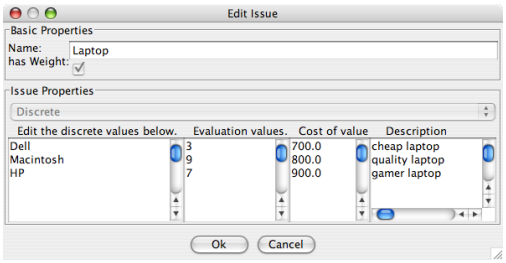

Figure 7: The discrete-issue editor after entering the evaluation values.

If you are satisfied with the evaluation values, press the Ok button. If you entered an illegal value, a warning will appear and the evaluation values will not be changed. Repeat step 4 for the other issues, until all evaluation values of all issues have been set.

3.5 Set the Issue Weights

As a final step to tune your utility space, you can adjust the relative weights of the issues, by using the sliders next to that issue. You can lock the weight value of an issue by clicking the checkbox next to the slider. The sum of the weights is automatically kept at 1, causing all unlocked sliders to change when you drag one of them.

3.6 Save your Utility Space

Once you have set all sliders and have filled out the evaluations for the options under each issue, select File > Save or click the save button, and save your file. It is possible to save incomplete utility space, for later completion although this is not recommended. Use an appropriate non-existing filename that refers to the domain it is related to, and makes clear it is a utility space file, e.g. laptop_buyer_utility.xml. The preference profile is automatically added to the repository under the domain.

4 Running Negotiations

To run a negotiation session, select File > New > Negotiation Session. The Session tab pops up (Figure 8). You need to set up the following items:

- The negotiation protocol. Only the Alternating offers is possible at this time. You can also use "Alternating Offers with separate deadlines (deprecated)" protocol for backwards compatibility with agents written for older versions of GENIUS.

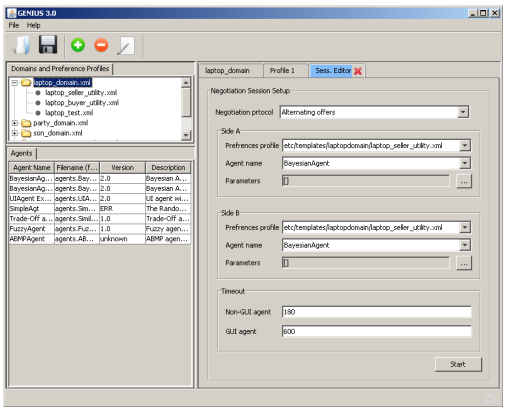

Figure 8: The negotiation set-up screen.

- Agent A/B preferences profile: the xml file containing the utility space of agent A/B. Usually this is a file with a name like laptop_A/B_utility.xml.
- Agent A/B name: the name of the negotiation agent that party A/B in the negotiation wants to use. You can use the provided agents, or use your own agent placed in the directory containing the `negotiator.jar` file. UIAgent allows you to manually control party A/B in the negotiation. SimpleAgent is an agent that automatically handles the negotiation for party A/B.

After these fields have been set appropriately, you can press the Start button to run the negotiation sessions.

4.1 Using an Automatic Agent

If you selected an automatic negotiation agent, for instance SimpleAgent, there will not appear any window while that agent has the turn. The agent should pose its bid within a reasonable time. After the agent made its bid, the other agent is given the turn. If both agents are automatic, no windows will appear at all during the entire negotiation, and only the output windows will show the ongoing negotiation results.

4.2 Negotiation Results

There are two types of results: the utility plot and the utility of the last bid for each round. Both are shown in the Progress tab, as displayed in Figure 9.

4.2.1 The Utility Plot

A utility plot will be provided during each negotiation. The plot shows points corresponding to all possible bids, the Pareto Frontier, Nash and Kalai points, all bids made by both agents and the final agreement.

4.2.2 Exchanged Offers

The results of the negotiation are shown in the Exchanged Offers part. This shows four columns: the session number, the final utility for agent A and B, and the Protol ErrorRemarks column. The utilities are the utilities of the final, accepted bid, as measured in the utility spaces of agent A and B (see the section Utility of a Bid).

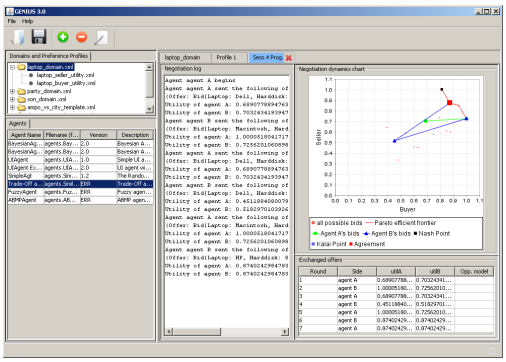

Figure 9: Progress tab. The utility plot also shows the Pareto Frontier (the red line), and the bidding sequence of agent A and agent B (the green and the blue lines).

4.3 Running a Tournament

Besides running a single negotiation session, it is also possible to run a tournament. In a tournament, a negotiation session is held for every possible combination of chosen agents and preference profiles for each side. To do this, select File > New > Tournament. The Tournament tab will appear as displayed in Figure 10.

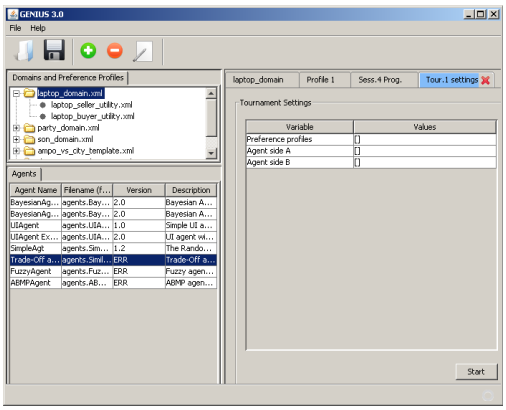

Figure 10: Tournament tab.

Double-click Preference profiles and select at least two profiles from the window that pops up (Figure 11). For every chosen domain, at least two profiles should be chosen. For example, choose laptop_seller_utility.xml and laptop_buyer_utility.xml.

Next, double-click Agent side A and select one or more agents from the window that pops up. Do the same for Agent side B and press the Start button. A Progress tab will appear as displayed in Figure 12. This tab contains the same information as a Progress tab for a single negotiation session, but additionally it has a tournament overview. If you click on a session in this overview, the details of that session will be displayed.

5 Writing a Negotiation Agent

This section discusses how you create an automatic negotiation agent. To explain this, the SimpleAgent.java code as provided in the installation package will be discussed. It is assumed that you are familiar with programming in Java. In case you need more information about JAVA programming, please use the following

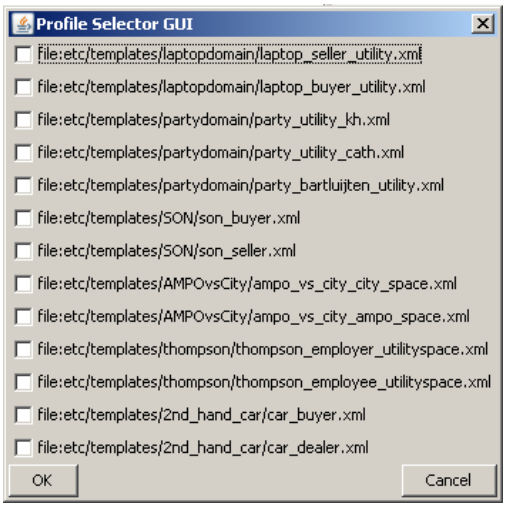

Figure 11: Profile Selector.

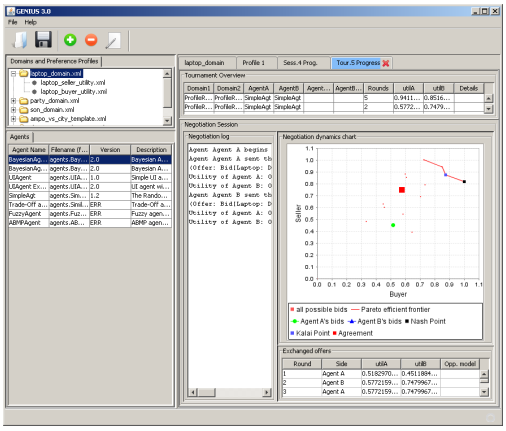

Figure 12: Progress tab for a tournament.

link: `http://java.sun.com/docs/books/tutorial/index.html`. The Java API definitions can be found on `http://java.sun.com/j2se/1.5.0/docs/api/index.html`. In the first place, a negotiation agent has to extend the negotiator.agents.Agent class. Table 1 shows the most important fields and methods of this class. For more information, please refer to the javadoc of GENIUS.

By extending Agent, your agent can access the fields as it likes. To implement your agent, you have to override two or three three classes:

- `public void ReceiveMessage(Action opponentAction)`
- `public Action chooseAction()`
- `public void init ()`

5.1 ReceiveMessage

The `ReceiveMessage(Action opponentAction)` informs you that the opponent just did `opponentAction`. The `opponentAction` may be null if you are the first to place a bid, or an `Offer` containing the bid of the

`UtilitySpace utilitySpace` The current utility space. This usually is an instance of the `UtilitySpace` that you specified in the negotiation set-up screen (Figure 8).
`public Timeline timeline` Use timeline for everything time-related, such as `getTime()`. Other time-related fields are deprecated. See the `Timeline` class for more details.
`public void sleep(double fraction)` Let the agent wait. Example: `sleep(0.1)` will let the agent sleep for 10% of the negotiation time (as defined by the Timeline).
`public double getUtility(Bid bid)` A convenience method to get the utility of a bid. This method will take discount factors into account, using the status of the current timeline.
`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. This function is called immediately after a ReceiveMessage, and only if the opponent made an Offer or if this is the first round in the session.
`String getName()` Returns the name of the agent. Please override this to give a proper name to your agent.

Table 1: The most important methods and fields of the Agent class.

opponent. It may also be an `Accept` or `EndNegotiation` action. The `chooseAction()` asks you to return an `Action` to make the next step in the negotiation.

In the SimpleAgent code, the following code is available for `ReceiveMessage` (Figure 13). This will be the typical code for automatic negotiation agents.

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

Figure 13: Example code for `ReceiveMessage`.

5.2 ChooseAction

Figure 14 shows the example code for the `chooseAction` method. 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 (see Negotiation Protocol) and results in a score of 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.

The Paccept function is a probabilistic acceptance function where P equals:

(5)

where u is the utility of the bid made by the opponent (as measured in our utility space), and t is the current

```
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());
                }
                sleep(0.005); // just for fun
        } catch (Exception e) {
                System.out.println("Exception in ChooseAction:"+e.getMessage());
                action=new Accept(getAgentID()); // best guess if things go wrong.
        }
        return action;
}
```

Figure 14: Example code for `chooseAction`.

time as a fraction of the total available time. Figure 15 shows how this function behaves depending on the utility and remaining time. It can take quite some time to figure out a formula that suits the requirements, but it is at the heart of the example agent.

The UtilitySpace and its functions should be considered as 'given', You may want to override the `init` function, to catch the `sessionNumber` and `sessionTotalNumber`. See the `SimpleAgent.java` example file. 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.

5.3 Compiling an Agent

To compile the agent, you put `YourAgent.java` code in the directory containing the `negotiator.jar` file, and use the command line

```
javac -cp negosimulator.jar YourAgent.java
```

After compilation, the resulting `YourAgent.class` file can be loaded into the negotiator simulator by typing "YourAgent" (fill in your actual agent name) in the negotiation set-up screen (Figure 8). Make sure your agent can be found in the same directory structure as is indicated by its package header.

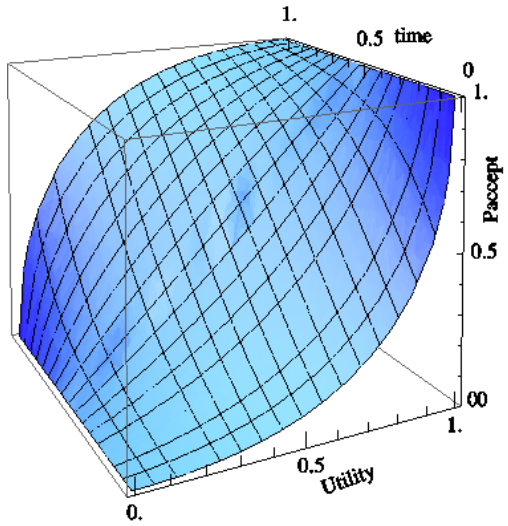

Figure 15: Paccept value as function of the utility and time (as a fraction of the total available time).

6 Data structures

For the documentation of the data structures that are relevant when writing a negotiation agent, please refer to the javadoc that can be found in your download of GENIUS.

7 Conclusions

Any comments and suggestions on the negotiation system and manuals can be mailed to `ai@mmi.tudelft.nl`.

8 References

[**sun07**] Sun Developer Network, JDK 6 Update 3. `http://java.sun.com/javase/downloads/index.jsp`