Proceedings of the Federated Conference on
Computer Science and Information Systems pp. 589–596

ISBN 978-83-60810-22-4

Developing intelligent bots for the Diplomacy game

Sylwia Polberg
Warsaw University of Technology
Email: sylwia.polberg@gmail.com

Marcin Paprzycki, Maria Ganzha
Polish Academy of Sciences
Email: firstname.lastname@ibspan.waw.pl

***Abstract*—This paper describes the design of an architecture of a bot capable of playing the Diplomacy game, to be used within the *dip* framework—a testbed for multi-agent negotiations. The proposed *SillyNegoBot*, is an extension of the *SillyBot*. It is designed to be used in the level-1 negotiations (as defined within the *dip* framework) taking place during the Diplomacy game.**

I. Introduction; the Diplomacy Game

THE DIPLOMACY board game was created in 1954 by Allan B. Callhamer. The game takes players back to Europe from the beginning of the 20th century. The aim of each player is to eliminate opponents and gain control over the continent, by any means necessary. To gain more influence, players can negotiate, create alliances, lie, break alliances and/or promises, etc. Outcome of the game depends only on players' decisions and behavior (there is no element of chance). More information about the game can be found in the Wikipedia [1], the Diplomacy Archive [2], and the rule book [3].

Since the game depends only on strategy and negotiations it became of interest to AI researchers. There were many projects that tried to create a successful *bot* for the Diplomacy game. References to most of them are available on the DAIDE project website [4]. Unfortunately, most of them are currently halted, or already dead [5]. This is often not only because the topic is hard, but also because it requires knowledge outside of computer science. Moreover, in many cases source codes of bots were not made available, and hence the total contribution to the state of knowledge was relatively small. Currently, the most active project is pursued in the Spanish Artificial Intelligence Research Institute (IIIA). This project aims not only at developing the *dip* framework (an environment, in which agents will be able to compete against each other and humans), but also to develop negotiating bots (however, these are not available yet).

The aim of this contribution is to summarize the rationale behind and describe the architecture of the *SillyNegoBot*, which is to be capable of level-1 negotiations (within the *dip* framework). The *SillyNegoBot* is an extension of the *dip* 0-level Diplomacy playing *SillyBot* (see, [6]).

A. Rules of the Diplomacy game

Let us start by briefly describing the rules of the Diplomacy game (for further details, see the rule book [3]). In order to explain how the game proceeds, we need to first introduce some basic notions. In the standard game there are 7 powers (players) — Austria, England, France, Germany, Italy, Russia and Turkey. The map of Europe is divided into 56 land and 19 sea provinces. Each province is further split into a certain number of coastal, sea and land regions. Among provinces, 34 are marked as Supply Centers. Controlling them allows a player to expand his army. Homes are supply centers that each player receives at the very start of the game. Every power has 3 of them, apart from Russia—in exchange for the longest battle front it controls 4 home provinces. Winning condition of the game is the control of at least 17 supply centers.

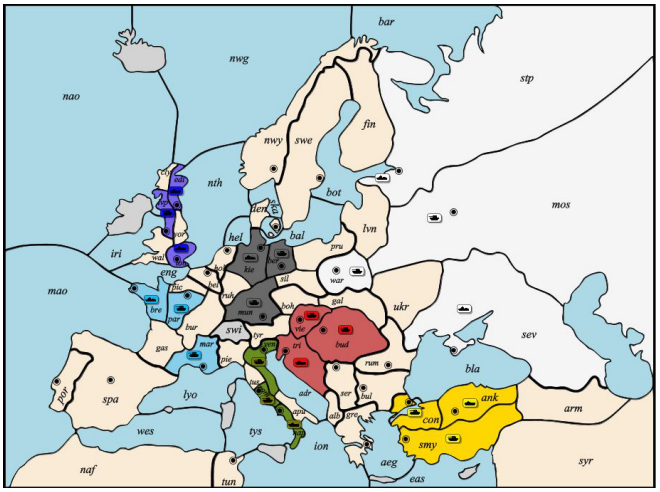

Fig. 1. Sample initial map from the dipgame [7].

B. Orders

During each phase of the game (see, below), an order has to be issued for each unit. Depending on the specific phase, an order can be either of the ones presented on Figure 2.

Both, in the case of a movement and attacking, unit with bigger support "wins" and moves to the desired region (captures it). When the support size is equal, units do not move. Interestingly, most of known diplomacy bots do not implement the convoy order. Note that the game-server executes orders

Order	Abbr.	Description
Hold	*HLD*	unit stays where it is
Move	*MTO*	units moves to an adjacent (free) region, or attacks it
Support	*SUP (supporting holding unit)* *SUPMTO (supporting moving unit)*	for unit *A* to support a stationary unit *B*, they have to be neighbors if unit *B* is moving, for *A* to support it, *A* has to reside next to *B*'s destination
Convoy	*CVY*	fleet order used to move an army from one coast to another
Retreat	*RTO*	move a unit that needs to be evacuated (escape) to a (free) adjacent region
Dislodge (disband)	*DSB*	used if a *Retreat* is not possible (dislodges the unit from the board)
Build	*BLD*	build a unit; if there are more resource centers than units, and the home province is free
Waive	*WVE*	sent if it is impossible, or undesired, to build a unit
Remove	*REM*	if there are more units than source centers player chooses unit to remove

Fig. 2. List of Diplomacy orders.

"simultaneously." In other words, time of their arrival is of no importance to their results.

1) Game play: Game proceeds in turns (representing years). Each turn is separated into five phases that differ in purpose and possible orders; in summary:

- *Spring*
  The first movement season. Orders *MTO*, *HLD*, *SUP* and *CVY* can be issued. During this phase, one usually moves units to areas (s)he wants to annex soon, or build up defenses against incoming enemies.
- *Spring retreats*
  Only *RTO* and *DSB* orders can be issued. If during the Spring phase a units is attacked and loses the fight (i.e. the attacker had greater support), this unit has to move out of the region. If there are no such regions, the *RTO* order has to be issued. Otherwise, given army or fleet is automatically dislodged from the board.
- *Fall*
  Season very similar to the Spring. The only difference is that after it ends, "newly" occupied supply centers are being annexed and become usable in the Winter phase.
- *Fall retreats*
  Exactly as Spring retreats.
- *Winter*
  At the end of the year players can expand their army (or fleet). In order to issue a *BLD* order, three conditions have to be met:
  - Province one wants to build a unit in, has to be an unoccupied home.
  - Province has to contain a region compliant with the type of unit to be build.
  - Amount of controlled supply centers has to be greater than the number of owned units.

  Usually one can build at most 3 units during one Winter—reason is obvious, there are only 3 homes (4 for Russia). Should a player have more units than supply centers, (s)he has to remove some of them in order to restore the balance (the *REM* order is issued for specific units). When no removal is needed or no builds are possible/desired, the *WVE* order is issued.

II. The *dip* Framework

In 2009, A. Fabregues and C. Sierra created the *dip* framework [8], which allows one to create Diplomacy playing bots and test their abilities against other bots (and humans). All libraries needed to write dip-bots can be found within the dip website [7]. The dip framework uses the *dip language* [8]. Currently, out of its 10 levels, we are interested in level-0 (*Order Issuing*), and level-1 (*Negotiations*). In Figure 3 we summarize how the latter is structured. Interestingly, the dip language not only differs from the earlier DAIDE language, but there is no 1-to-1 relation between them. This difference does not affect the game rules—only the level 0 communication is defined in the rulebook [3], while negotiations are independent and are not defined in the original board game. During negotiations, agents make proposals to other agents,

$$
\begin{aligned}
\pounds_1 ::= & \text{propose}(\alpha,\beta,\text{deal})|\text{accept}(\alpha,\beta,\text{deal})|\text{reject}(\alpha,\beta,\text{deal})| \\
& \text{withdraw}(\alpha,\beta) \\
& \text{deal} ::= \text{Commit}(\alpha,\beta,\varphi)^+|\text{Agree}(\beta,\varphi) \\
\varphi ::= & \underline{\text{predicate}}|\text{Do}(\alpha,\underline{\text{action}})|\varphi\wedge\varphi|\neg\varphi \\
\beta ::= & \alpha^+ \\
\alpha ::= & \underline{agent}
\end{aligned}
$$

Fig. 3. dip level-1 language.

and accept or reject ones they receive. In order to end a negotiation (message exchange) with a given power, an agent sends a withdraw message (similar to a "bye" after a finished chat). Note that this ends only a specific negotiation, (other negotiations between these agents may ensue later). Agents can "talk" about agreeing on truthfulness of some facts—such as keeping a peace, or committing to do something. By an action we understand the 0-level orders.

Note that the predicates used at the level-1 are typically limited to the following offers (and thus define the scope of our work):

- PCE(power+)—peace between a group of powers
- ALY(power+, power+)—alliance between a group of powers against some other group of powers.

III. SillyBot agent

Let us now briefly describe the SillyBot (a predecessor to the SillyNegoBot), which is capable of playing using the level 0 language. Out of many designs we have tested, this one proved to be the most successful one. For further details please refer to [6].

A. Bot design

Key elements of the design of the SillyBot were (see, also Figure 4:

- Phase Graders—responsible for placing *Requests*.
- Board Analyzer—consisting of Board Assistant, Union Manager and Threat Assistant.
- Request System—mechanism evaluating *Requests*.
- Heuristic—responsible for picking a *Request* solution, prioritizing *Requests*, and *Request* filtering.

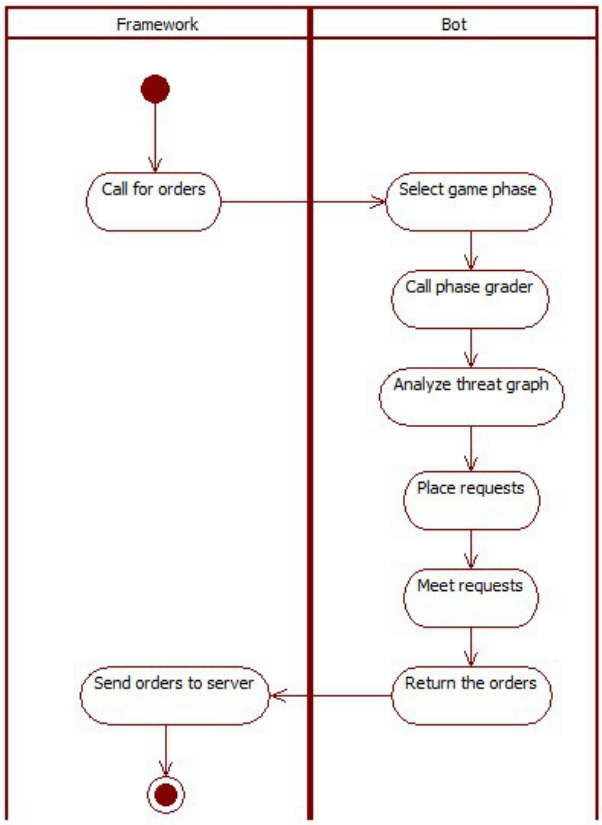

Fig. 4. Structure of the SillyBot agent.

To make the description more understandable, let us start from the *Request*, which is a simple entity that means "I want someone to :an order: in a given province." Each level 0 order has a respective *Request*. For example, let us assume that there is a province worthy annexing; we will want to issue an MTO order (for a specific unit). Therefore, we will place a MTO *Request* for that area. Moreover, each *Request* has to contain a list of possible units that can perform the desired order within a given province. When the SillyBot prepares actual orders to be issued, the *Graders* generate *Requests*, which are collected and sent to the *Request Meeter* (it is an element of the system responsible for satisfying *Requests*).

To handle the *Request* evaluation, an algorithm based on a tree structure has been developed. Here, the root of the tree is empty. We take first *Request* and expand the root—each node represents an actual sequence of orders satisfying the *Request*. There can be many ways to fulfill a *Request*, hence multiple nodes. For example, if we want to occupy Paris and have armies in Burgundy and Gascony, our root will have two children—one containing MTO(X,BURAMY,PARAMY), and the other MTO(X,GASAMY,PARAMY). Order sequences with more than just a single order can occur only in the case of a SUP or SUPMTO *Requests*. This should be obvious, as we can need help of more than just one unit. Created nodes are later filtered for collisions—we will not want to have an order issued for a unit that was earlier assigned a different order. Another filter is to establish whether satisfying the *Request* with orders in the node will later give us the chance to find a node fulfilling the next *Request*. This basically means that if among a group of nodes some leave at least one option for the next *Request*, we will not bother with the rest and apply the heuristic only to them. To each remaining node layer, another heuristic is applied that attempts to pick the best node. Its focal point is how "shared" node is—e.g. how many other *Requests* the unit we have picked can satisfy. With already sorted and grouped *Requests*, we aim to fulfill as many as we can. Here, result matters most, the fact who will go where is of secondary importance. In the final stage of the algorithm, data from all chosen nodes is merged into a single order list and returned to the *Graders*. There are several *Graders*—one for the building phase, one for retreating phase and three for the movement phase. This is due to the fact that during the start, middle and end of the game we may want to apply different *Request* placing, as priorities can change.

Results of initial tests have been reported in [6]. They were promising enough to proceed with further developments.

IV. The SillyNegoBot

The SillyNegoBot consists of 9 agents (for the reason behind its structure, see below). It is built on top of the SillyBot, to handle the level 1 dip language. Specifically, the framework agent (further referred to as, the *Mother*) extends the SillyBot and uses its functionalities to handle the level 0 functions (recall that the language levels are inclusive—every higher level contains previous ones). It adds functions and classes necessary from the point of view of framework, in order to handle the technical side of negotiation messages. Moreover it contains means of communicating with the remaining 8

agents. Logically, the SillyNegoBot consists of the Emotional System and Rational System. Although they share multiple functionalities, they are kept separated in order to be turned on and off for testing (e.g. to represent the emotion—rationality conflict). The rationale for including emotions in the design of the SillyNegoBot has been discussed in detail in [5].

For the design and implementation of the SillyNegoBot, we have decided to use an already existing knowledge driven model—JADEX, that is based on the BDI approach [9]. As discussed in [5], we have decided to create a bot consisting of eight separate (sub)agents: the *Mother*, six *Ambassadors*, and an *Arbitrator*. In the current version we have extended the system by the *MotherMessenger* agent, responsible for transferring all level 1 reasoning to JADEX. This encapsulates the negotiation-oriented reasoning (thus allowing to change the negotiation "brain" without touching the remaining parts of the system). After discussing the purpose behind all agents we will describe the communication between them and the complete architecture of the SillyNegoBot (for more details, see, also [5]).

A. Agents

1) Mother: The *Mother* agent (a slightly modified SillyBot) has two main tasks—the game (dip) server communication, and specification of dip level 0 orders. Furthermore, it initializes the platform for other agents and launches the *MotherMessenger*. Finally, it controls passing messages between the *MotherMessenger* and the server.

2) MotherMessenger: This agent, upon receiving appropriate message from the *Mother*, launches all other agents. It is responsible for distributing messages, e.g. *Ambassador of Germany* might not be interested in messages meant for the *Ambassador of Italy*. At the end of the game it is responsible for platform shutdown.

3) Ambassadors: The six *Ambassadors* are the main "thinking" part of the system. Note that, using a single agent to handle all "reasoning," would result in a single, very large, belief base, while requiring only a part of it for each decision. Moreover, one would have to keep track of six communication "channels" at the same time (recall that there are six opposing powers). This would complicate the implementation, and could lead to decreased efficiency. Therefore, we decided to use six *Ambassadors* (JADEX agents; one per opponent). The *Ambassadors*, facilitate level 1 messaging, and reasoning based on their exchange (Negotiations). Note that, due to this separation, we have created a layer of negotiations internal to the bot; e.g. one *Ambassador* may need a piece of knowledge owned by another *Ambassador*. To handle such situation, and to deal with conflicting recommendations originating from independently working *Ambassadors*, we have created the *Arbitrator* agent.

4) Arbitrator: The six *Ambassadors* will sometimes require acceptation of a goal or an action. For example, when all (or at least some) *Ambassadors* decide to attack powers they are assigned to, a decision has to be made, which attack should materialize (while others are dropped or postponed). To deal with this problem we introduced another agent—an *Arbitrator*—responsible for mediating conflicting recommendations obtained from the *Ambassadors*. Obviously, this task could have been placed within the *Mother* agent (or the *MotherMessenger*), but this would result in putting too many unrelated functions within a single agent. Therefore, we have followed a "single agent per major functionality" paradigm of agent system design.

B. Architecture of the SillyNegoBot

Here. we start from key elements that are independent of the JADEX platform. Next, we describe how we use the BDI model. For further details concerning the reasoning behind each element, please refer to [5].

1) Messages: Messages passed between agents will be encapsulated using FIPA standards. Their mapping is presented in Figure 5. In the current version of the bot we have simplified some messages and used the functionality taken away from them to create new ones. Internal messages in the system can be of the following types:

- *START* (*START((POWER, AGENT), (POWER, AGENT),...)*)– sent by the *MotherMessenger* to all *Ambassadors* (as the first message). It and defines the power that the given *Ambassador* is responsible for. The *Ambassador* agent is supposed to extract its own assignment and remember the powers of others in order to make the internal communication easier.
- *INFO* (*INFO(ORDER), INFO(MESSAGE)...*)—exchanged between all agents in the bot to inform about data, e.g. the *MotherMessenger* agent passes an incoming message to the *Ambassador*. It can also be the case that an *Ambassador* sends to the *Mother* (through *MotherMessenger*) information as to who is allied with whom (level 0 requires such data for proper threat estimation).
- *PASS* (*PASS(MESSAGE)*)—used to ask the *MotherMessenger* and the *Mother* agents to pass messages from the *Ambassador* to the given power. Note that a direct communication is impossible without writing our own server.
- *FETCH* (*FETCH(POWER), FETCH(ORDER), FETCH(PREDICATE),...*—asks given agent to provide specific information (beliefs, emotions and so on) concerning a given opponent, relations between powers, given order or belief, feeling, etc.
- *ARB ID* (*ARB ID(AGENT)*)—contains the identifier of the *Arbitrator* agent.
- *BOARD* (*BOARD(GAME)*)—message sent from the *Mother* to the *MotherMessenger*. State of the board is read and transformed into *OWN* predicates, which are later passed to the *Ambassadors*. If it is received for the first time, Power information is read and used to launch other agents in the platform. It requires no FIPA encapsulation, as it sent through a socket.
- *END*—end of the game, shut the platform down.

- *NEXTPHASE* (*NEXTPHASE(TIMELINE)*)—message marking a new phase in the game.

Internal message	FIPA encapsulation
START	INFORM
INFO	INFORM
PASS	INFORM
FETCH	REQUEST
ARB ID	INFORM
END	INFORM
NEXTPHASE	INFORM

Fig. 5. FIPA encapsulation of internal messages.

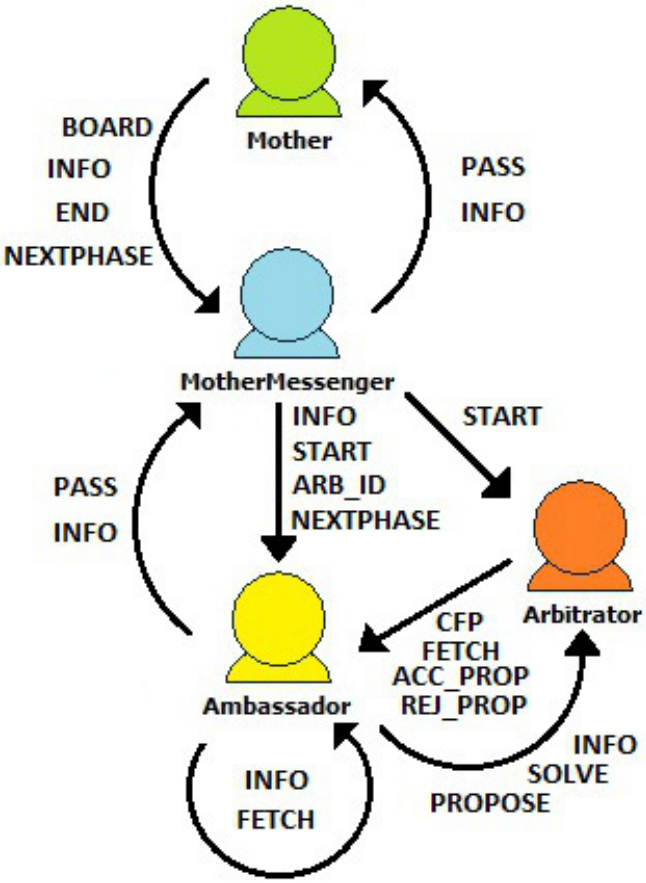

Fig. 6. Message flow diagram.

We drop the internal parsing in the following messages:

- *SOLVE* (*SOLVE(MESSAGES)*)—sent by the *Ambassador* to the *Arbitrator*, in order to receive acceptance of moves. In response the *Arbitrator* sends out a *CFP* message, which should be answered with *PROPOSE* messages that *Ambassadors* want to send to their respective powers. In other words—an initial message for conflict resolution.
- *ACCEPT PROPOSAL, REJECT PROPOSAL*—to be used as an answer for the *PROPOSE* message.

2) Emotions: Emotions impact the probability of other powers actions—how will they react to negotiations (offers and counter-offers), whom will they help, and who not. Simple analysis of "what action benefits one most" is not enough in the case of a game like Diplomacy, and often fails to lead to the optimal strategy (in particular, in case of bots playing against humans). Typical primary emotions include fear, sadness, anger and happiness [10]. There is quite a number of models allowing introducing emotions into a system like ours, some more complex than others. When it comes to computers, HUMAINE Emotion Annotation and Representation Language EARL is worth mentioning [11]. However, it distinguishes a total of 48 emotions, out of which some are of no interest to us. Moreover attempting at using it would move the project to a completely new complexity level, and change the focus too far into the human emotions. The level 5 dip language provides us with the following emotions: *Very Happy*, *Happy*, *Sad* and *Angry*, which is partially related to the theoretical models (e.g. the James model, the Weiner and Graham, and the main categories of the Parrot model [10]). However, fear is not included in our set—basically because it is a weakness when felt by our bot, and a fact impossible to verify when felt by other players (see, also [5]).

From basic emotions we create two aggregated values, *Like*, and *Emotional Trust*. The first one represents the general outcome and how we feel about the opponent, while the latter represents the comfort in/confidence towards someone.

Emotion facts are stored as triples *Power—Feeling—Reason*, where *Power* represents who "felt," *Feeling* can be: *Very Happy*, *Happy*, *Sad* and *Angry*, while *Reason* is a list of *Expressions* that caused given emotion to arise—basically it is the left hand side of rules that are responsible for creating this fact and perhaps increasing its probability. For example "Germany is happy with Italy offering peace" will be represented as GERMANY—HAPPY—PROPOSE(ITA, GER, COMMIT(ITA, GER, PCE(ITA,GER))). Basic emotions used in the dip framework are extended by two new numerical variables—*Liking* and *Emotional Trust*—that represent how "in general" we feel about our opponent.

3) Personality: Players differ—some cheat and lie, some don't. Some forgive and forget, while others hold grudges till the end of the game (and sometimes, beyond). They can like some actions more than others. This all calls for creating a structure that will represent such characteristics of a bot, its "personality." In our approach we express personalities in a *.pers* file that is loaded into a belief set. Note that they impact almost every process in the bot—from creating conclusions, relationships, to decision making. Personality contains also additional information—how emotionally we react to some actions, whether some we despise more than others. Therefore, we can model how *Like* and *Emotional Trust* are decreased or increased, based on events in the game. Here, we present sample characteristics:

- Boolean canLie.
- List ⟨int⟩ initialLike—initial value of like for opponents.
- List ⟨int⟩ likeWeight—here, we can define how much

weight is put to different types of actions or emotions; this is useful for expressing a person that "only looks at good sides," that gains "liking" fast when made *Happy*, but loses it slowly when made *Sad* or *Angry*, etc. Same applies to changes of opinion about a player that attacks him or supports him.
- int *likeLimit*—value of liking below which we stop negotiation ("I hate you, I'm not talking to you").

All entries are in a form *Class—Name—Values*. One line is one such triple, with elements separated with the — character. Thanks to this we can easily use Java Reflection to parse the file and load all the necessary data. Here is a fagment of the .pers file:

- Boolean—canLie—0
- Vector ⟨Integer⟩—initialLike—50 50 50 50 50 50
- int—memlevels—5
- Vector ⟨Integer⟩—memvals—10 9 7 6 4

4) Reasoner: Our bot is capable of creating conclusions from facts thanks to a rule based reasoner. Based on the rules defined in an external .rf file, our agents can create new facts. Rules are in a form $\{Expression+\} \rightarrow \{Expression+\}|p$, where the Expression is a helper class containing either an order, a message, a predicate, or a function. p stands for initial probability that the drawn conclusion is correct. Some rules might have 100% probability. In other cases, we treat them as some initial data that can undergo many changes during computations. Here we can see the predicates connected to the level-0 language, and where do they come from:

- ATK(unit owner, region owner)—attack—when a destination of MTO(unit, region) is occupied by opponent's unit, or it is a supply center controlled by an opponent.
- DEF(power init, power aim)—aim power defended from init power (symmetric to ATK).
- PTAH(unit owner, power,x)—in position to attack/help in x rounds—currently distance x is set to not bigger than 2, can be generated by MTO, HLD, RTO and BLD orders. This value was found empirically as being not too far for conclusion to be insignificant, and not too close to trigger the reasoning too late.
- HLP(unit owner, help receiver)—if an opponent issued a support order towards rival's unit, we can say it was *helping* it. Generated by SUP(unit, HLD(unit)) and SUP(unit,MTO(unit,region)).
- DSTR(power,power)—a power *destroyed* opponent's unit—generated directly by forced DSB(unit), or indirectly by REM(unit) by taking away resources.
- STR(power,power)— one power is *stronger* than the other—comes from calculating resources and units.
- OWN(power, region)—power owns given region—created from the current board state.
- NONE(power, power)—nothing occurred between two powers.

For level 1 following predicates were introduced:

- WAR(Power, Power)—given powers are at war—can be generated for several reasons, e.g. attack, helping the attacker, being allied with the attacker.
- ALY(Power, Power+)—given powers are allied/at peace—can be caused by received messages, observed help or affiliation.
- LIKE(Power, Power)—needs to be used with the EVAL function, stands for *Like*.
- ETRUST(Power, Power)—needs to be used with the EVAL function, stands for *Emotional Trust*.
- TRUST(Power, Power)—needs to be used with the EVAL function, stands for *Trust*.
- REP(Power)—needs to be used with the EVAL function, stands for *Reputation*.
- EREP(Power)—needs to be used with the EVAL function, stands for *Emotional Reputation*.
- HAPPY(Power, Power, expression), VHAPPY(Power, Power, expression), ANGRY(Power, Power, expression), SAD(Power, Power, expression)—used to express how given power felt about opponents action/sent message/etc.

We also have the five functions used for the *Time line* related analysis:

- FUT(expression)—reasoning about future.
- PAST(expression)—reasoning about past.
- INC(expression)—increase value by a modifier defined in the .pers file.
- DEC(expression)—decrease value by a modifier defined in the .pers file.
- EVAL(expression)—used for emotion evaluation, it fetches the numerical value of given emotion.

As stated above, beliefs contain a variable *Time line* and can be grouped by it. To switch between such groups we use the FUT and PAST functions. We use such to express opinions like "Opponents units are close to me, he might attack me in the next round". A simple example of when can that be useful—"German units are getting close to the Italian units, and hence Germany might attack Italy", represented as $PTAH(GER, ITA, 2) \rightarrow FUT(ATK(GER, ITA))|30$ (value 30% is just a sample probability).

The *INC* and *DEC* functions are used to deal with numerical variables such as *Trust*, *Reputation*, their emotional equivalents, and *Liking*. They provide a signal that a value should be decreased or increased, e.g. "Germany attacked Italy hence Italy likes Germany less" expressed as: $ATK(GER, ITA) \rightarrow DEC(LIKE(ITA, GER))|100$. The quantity by which we should increase or decrease the value is defined in the *Personality* of the bot and can be further affected by the current situation; e.g. if we are close to losing, we might care less or care more about specific situations surrounding us.

The general structure of the *Reasoner* is depicted in Figure 7. As it can be easily seen, the .rf file is first analyzed by a *Rule Reader*, based on a tree parser (thanks to it we can handle any level of nesting). The *Rules* are then passed to the *Rule Manager* that handles firing, matching and substitution. Whenever the belief base is extended, we fire the rules, generate all possible facts and add them to the belief base.

Algorithm IV.1: REASONER($newbelief$)

$FireRules(newbelief)$;
$PerformMatching()$;
$List < Expression > out = Substitution()$;
if $out! = null$
  then $GenerateNewBeliefs(out)$;

Fig. 7. Structure of the Reasoner.

Rule matching is performed as follows: we read all variables in the rule premises, and all variables in the beliefs sequence, and we check them for application. If we managed to assign them in a one-to-one manner, such that the order is preserved, it means that we can draw conclusions. Variables in the created expressions, such as Power x, are substituted by real values (e.g. France, Russia), and finished elements can be added into the belief base. Thanks to the Java Reflection, we can easily extend predicate, function, etc., without any further need of modifying the mechanism. For an example illustrating how the reasoner works, please refer to [5].

Here is an extract from the .rf file:

- MTOOrder(Power x, Region unit, Region dest) + OWN(Power y, Region dest) = ATK(Power x, Power y) — 100
- ATK(Power x, Power y) = WAR(Power x, Power y) — 100
- DSTR(Power x, Power y) = WAR(Power x, Power y) — 100
- WAR(Power x, Power z) + ALY(Power y, Power x) = WAR(Power y, Power x) — 60

Please note that here "+" and "=" do not stand for arithmetic operations. They are simply used to express "and" and implication. It is just a personal choice of symbols.

5) Making a decision: The *Ambassador* and the *Arbitrator* choose which recommendations to follow and which should be (at least temporarily) abandoned, in the following way:

- Compute all consequences—this basically means that using the reasoner we estimate the effects of our decisions, e.g. taking into account consequences for the owned land, feelings of our opponents, possible opponent actions, etc.
- Compute likelihood of given outcome—is evaluated on the basis of probabilities of specific beliefs that form it.
- Filtering—we remove decisions that would lead us to something we cannot, or do not want to, do, e.g. betrayal (breaking a promise, or attacking an ally without prior warning)
- Evaluate the benefits of decisions—whether we get some land, if yes, how good it is; do we make enemies, if yes, how bad it is; do we do something we like and approve, if yes, how much; and so on. Very important is the fact whether our decision gets us any closer to our goals.

The outcome of the algorithm is an assignment of a numerical value to a decision. This allows us to rank possibilities and as a result the highest ranked one is the winner. However, we send to the *Arbitrator* three best results, just in the case it rejects a choice we established to be the best (from our perspective).

The decision cycle of the *Arbitrator* is very similar when it comes to computing consequences and judging them. However, its main aim is to agree to the most beneficial combination of *Ambassadors'* proposals. In this way, we can say that while *Ambassador* think locally, the *Arbitrator* thinks globally.

6) Embedding it all into BDI: All knowledge our bot has—Personality, power assignments, facts etc., is held in the belief base. It consists of four main belief sets, all represented as JADEX Tuples:

- Assignment Belief Set—stored as pairs *Agent—Power*
- Setup Belief Set—holds personality elements and technical information for the bot
- Emotion Fact Belief Set—holds triples *Time line—Emotional Fact—Memory time*. *Time line* is the time given belief was generated, *Memory time* is the number of rounds it should be stored. The *Emotional fact* is an emotional predicate (possibly combined with functional) such as DEC(LIKE(GER, ITA)). It is accompanied by the reason for generation.
- Rational Fact Belief Set—as above, for rational facts.
- Relationship Belief Set—stores powers and their relationships with each other: *Trust*, *Reputation*, *Emotional Reputation*, *Like*. Note that it was separated from other belief sets, in order to reduce the complexity of computation—we are going to use them often, and it is handy to save them separately, rather than having to repeatedly search through the belief set.

In our bot, we make an extensive use of triggers and goals. The first category is the events caused by an incoming message—they are used to launch the FIPA Arbitrator/ Ambassador message plans (JADEX plans that are meant to react to messages). Next, content is read and an appropriate event dispatched. It fires the internal message plan that performs the desired action—what interests us most at this point, is the INFO received from the *MotherMessenger*, and what is happening in the *Arbitrator* during the decision agreement. Other messages mainly operate on the belief base—add, remove, fetch, etc. Manipulating beliefs can fire the reasoner plan (JADEX plan that launches the reasoner and adds created beliefs to the belief base) that is meant to draw all possible conclusions from the provided data. Message sending is fired with a send event, dispatched from any plan. The schema of message exchange between the *MotherMessenger* and the *Arbitrator* is represented in Figure 8.

After receiving a negotiation message passed by the *Mother* and the *Mother Messenger*, we can finally start thinking and planning. As mentioned before, we evaluate the consequences. In the meantime it might happen that one of the JADEX agents will demand additional information or some confirmation from another agent. We "pause" the fire plans, mentioned above, and wait for an answer (for the time defined in the *Setup Belief Set*). After deciding what to do, we hold 3 best possible choices and call the *Arbitrator* for an approval, as

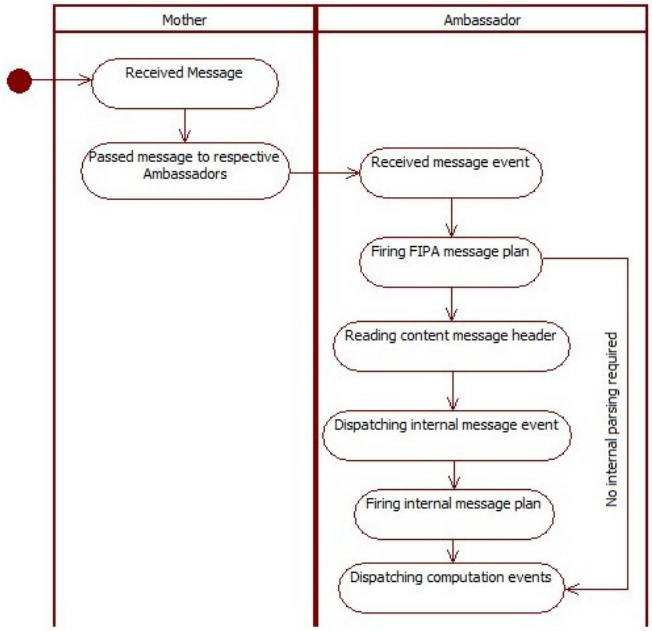

Fig. 8. Recceiving messages diagram.

depicted in Figure 9. The received answer is then passed to the *MotherMessenger* agent.

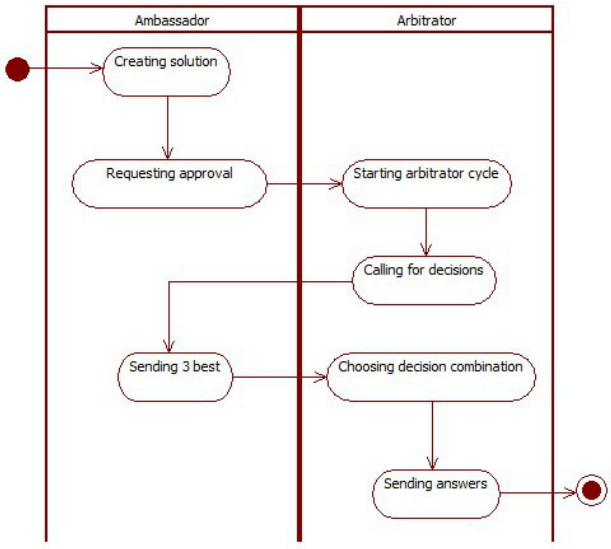

Fig. 9. Arbitrator cycle.

However, with the current plans and events, we have a problem—the first act of negotiation from the *Ambassadors* side can start only as an effect of a received message. It means that we need a plan that is "always running in the background," and performing part of the computations mentioned above to check whether we need some interaction with opponents. If such a need is established, we create an appropriate goal and fire the grading plan that will choose what should be done. This is the default behavior. Of course, it can happen that the previous step will precisely know what is required and pass such data to the *Ambassador*. For technical issues, such as how precisely the triggers work, see [9].

V. Concluding remarks and preliminary tests

In this paper we have described the background of the Diplomacy game and the architecture of the bot that is to play it using Negotiations. After correcting some minor implementation errors, the bot will be tested against humans, and the IIIA—CSIC bot. One of the important directions of experimental testing of the proposed design will be use of different personality setups. Most interesting should be the tests with different emotional – rational ratios, e.g. 0:100, 50:50, 100:0. Finally, we expect to use the experimental data to tune up the bot's heuristic and in this way complete the work with dip level-1.

Initial tests with the 50:50 ratio included 10 games with 3 human players (3, 3 and 4 games), in proportion 1 SillyNegoBot, 6 DumbBots and 1 human. Such choice was made to analyze the behavior of the SillyNegoBot. We used DumbBots to have clear view at the SillyNegoBot—human negotiations. Out of the conducted games, 3 ended prematurely due to technical errors, out of the remaining 7, our bot won 4. Results gave us more information on what needs to be corrected both from the implementation and the behavior side. However, the overall conclusion is that the current version of the SillyNegoBot is too trustful and too nice towards other players.

Acknowledgment

We would like to thank A. Fabregues and C. Sierra for support with the framework and the bot, and the COST Action IC0801 for funding the STSM visit that made creation of the architecture design possible.

References

[1] Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Diplomacy_(game).
[2] Diplomacy Archive http://www.diplom.org/~diparch/god.htm
[3] Calhamer, Allan B. The Rules of Diplomacy 4th Edition. s.l. : Avalon Hill Game Co., 2000.
[4] Diplomacy AI Centre. http://www.daide.org.uk/l.
[5] Sylwia Polberg, SillyNegoBot Architecture, to appear in: M. Essaaidi, M. Ganzha, M. Paprzycki (eds.), Software Agents, Agent Systems and Applications, IOS Press, 2011
[6] Sylwia Polberg, Shupantha Kazi Imam, Developing Bots for the Diplomacy Game, submitted for publication.
[7] Fabregues, Angela and Sierra, Carles. Dipgame. www.dipgame.org.
[8] Fabregues, Angela and Sierra, Carles. Testbed for Multiagent Systems. http://www.iiia.csic.es/files/pdfs/DiplomacyTestBed.pdf.
[9] Braubach, Lars, et al. JADEX - BDI Agent System. http://jadex-agents.informatik.uni-hamburg.de/xwiki/bin/view/About/Overview.
[10] Emotions http://changingminds.org/explanations/emotions/primary_secondary.htm http://changingminds.org/explanations/emotions/basic\%20emotions.htm
[11] EARL http://emotion-research.net/projects/humaine/earl