Using JaCa technologies for a multi-agent based Risk bot

Arcaroli Cristian, Lazzarini Marco, Molari Federico

Seconda facoltà di Ingegneria
Università di Bologna

17th May 2011

Abstract

Our work it's intended to develop a smart agent-based bot capable of doing autonomously and smartly the large amount of chooses that the Risk game requires. Specifically, multi-agent architecture provides a service, exploited by bot to execute the different game phases and take decisions accordingly to the particular game situation.

The bot is developed following the structure given by Sillysoft, producer of the platform LuxDelux [6], that provides the game environment and Risk constraints.

With reference to Risk Wikipedia page [7], *"Risk is a strategic board game. It's a turn-based game for two to six players. The standard version is played on a board depicting a political map of the Earth, divided into forty-two territories, which are grouped into six continents. The primary object of the game is "world domination," or "to occupy every territory on the board and in so doing, eliminate all other players."Players control armies with which they attempt to capture territories from other players, with results determined by dice rolls."*

Multi-agent system for bot's reasoning is realized integrating functionalities of Jason [2] and CArtAgO [1]. The integration of the two technologies allows the creation of the environment for the agent-based decision system. Moreover, CArtAgO implements the interface between the game and the MAS, interpreting the role of bridge. Strategies implementation follows a thesis work, MARS [4], that exploits multi-agent architecture for the creation of a Risk bot. This structure was freely extended and revised, for the creation of a new bot called Attila.

Contents

1 Risk model

In this section will be presented a brief description of the game *Risk*, focusing on the features of the various game phases and the constraints in the evolution of the game.

1.1 General game description

Risk is a board game, from 2 to 6 players, with the goal, for the players, to conquer the entire world.
Risk is composed by following items:

- A *board*, representing a simplified world map including 42 territories divided into 6 continents: Africa, Asia, Australia, Europe, North America and South America.
- A *deck of cards*, where each card refers to a specific territory and has a symbol (cannon, horseman or foot soldier); there are also two special cards, called jokers, that refer no territories and have a special behavior (see relative paragraph in section 1.2).
- *5 dice*, for the attacking phase, where 3 of them are for the attacker and 2 are for the defender.
- *Armies*, the units placeable on the board; each of them has the same value, and can move only between neighbor territories.

After an initial setup phase, wherein each player gets his initial territories and armies, the Risk match goes on by alternating players turns.
Each turn is composed by four subsequent phases:

- Trade in cards
- Place armies
- Attack
- Fortify

Turn phases are now described with higher level of details.

1.2 Phases description

Game setup This phase starts the Risk match, and will not be repeated during the match itself.
The initial amount of armies for each player depends on the number of the players: it goes from 40 armies each in a 2-players match to 22 armies each in a 6-players match.
The occupation of territories can be done in two different ways:

- every player occupies an unoccupied territory, until no more territories are free, then placing the remaining armies in the owned territories.
- every player gets by chance his part of the cards deck and place armies in the territories specified in owned cards.

Now, with a roll of a dice, players order is chosen and the game starts.

Trade in cards Every time a player conquers a territory, he gets a card from the deck. If a player has three cards with the same symbol or one of each, he can trades them in. A joker can represent any one of the three symbols.
Cards are used to get extra armies; the number of armies a player can obtain depends on the particular rule used in the game: in this version of Risk we used rule 5-5-5 (more similar to classic Italian *Risiko*), so every time a player trades in cards, he gets always five armies (differently from other rules in which the number of armies grows up after every trade).
For each traded card that refers a territory owned by player, he receives two more armies, with the constraint to place them in that territory.

Place armies At beginning of his turn, a player is able to increment the number of armies on the board. He gets more armies depending on three variables:

- Number of owned territories: a player receives a number of armies equals to number of owned territories divided by three, rounded to the lower integer value. Even if this value is lower than three, he receives three armies.
- Number of completely owned continents: if a player completely owns a continent, he receives a certain number of bonus armies; this number

is fixed for each continent: Asia has a bonus of 7 armies, Europe and North America 5, Africa 3, Australia and South America 2.

- Trade in cards.

Attack After armies placement, the player is able to attack enemy territories. The only constraint is that he can attack only from owned territories with more than one army. The result of the attack is given by dice.
The attacker choses the number of armies to attack with, and roll that number of dice (up to three dice to roll). The defender rolls a die for each army in the defending territory (up to two dice to roll).
The highest attacking die is matched with the highest defending die, the one with the lowest value loses an army. In case of a tie the attacker loses.
If the defender runs out the armies the attacker conquers the territory and can move his armies in the new territory. This number of armies can't be less than the number of dice rolled by the attacker in the last, winning attack.

Fortify Fortification phase ends every player turn. In this phase armies can be moved between owned neighbor territories to reinforce defense.
There are many rules in Risk, that allow different moving strategies in fortification; the one used in Risk is called *"common house rule"*: it says that any army can be moved to any neighbor territory, but for a total of one step during fortification phase, respecting the constraint that every owned territory must have always at least one army.

2 Problem Analysis

2.1 Architecture

Following the A&A (Agent & Artifact) meta-model [5], a multi-agent system can be modeled in terms of agents and artifacts (as the name suggests), and workspaces.
Agents encapsulate the reasoning activity that leads to a strategy definition for each game phase. Artifacts are passive components used by agents to interact with the environment (the Lux framework) and exploit external resources or functionalities (algorithms, tables...). Workspaces are the conceptual containers of agents and artifacts that define the topology of the

system.

2.1.1 Agent distribution

There are two main ways [3] to structure the multi-agent system.

- *Task-oriented*: agents are characterized by a specific relevant task (for example handling cards, choose in which territory place the armies, choose which territory attack...). Agents are heterogeneous and this make the system strongly non scalable with the dimension of the board (i.e. the number of territories).

- *Entity-oriented*: each agent represents an entity of a relevant type (for example territories or armies). Agents are homogeneous and this leads the system to be scalable with the dimension of the board.

We choose a entity-oriented agent system, with each agent representing a territory of the board. This solution is a good compromise between scalability and complexity of the system.
In order to coordinate all the territory agents and to facilitate the auction system between all possible choices proposed by each territory agent, we introduced a mediator agent responsible for this task. Since the mediator agent represents a single point for interaction between territory agents and the game framework, territory agents don't need to know about other territory agents work, making easier synchronization and the interaction itself.

2.1.2 Environment model

From the multi-agent system point of view the game framework is the environment in which the system works. It makes sense thinking about artifacts as the entities through which the agents can interact with the game framework. In order to reach this aim, we introduced a bridge artifact that can be used by the mediator agent to perceive which service the bot needs (specifically, for which phase of the game is requested a strategy) and to act directly on the game framework, executing the strategy previously established.
Moreover each agent needs to know the updated status of the game framework (in order to take sound decisions). We introduced a territory artifact that is updated at the beginning of each game phase. This artifact is observed by the agents to have reliable information on the game status.

Finally artifacts can be used to encapsulate predictable operations like algorithms that are too complex to be easily inserted in an agent. Hence we added two support artifacts that provide to the agents that need them the functionalities to evaluate a set of cards (a cards artifact) or to retrieve conflict odds (conflict artifact).

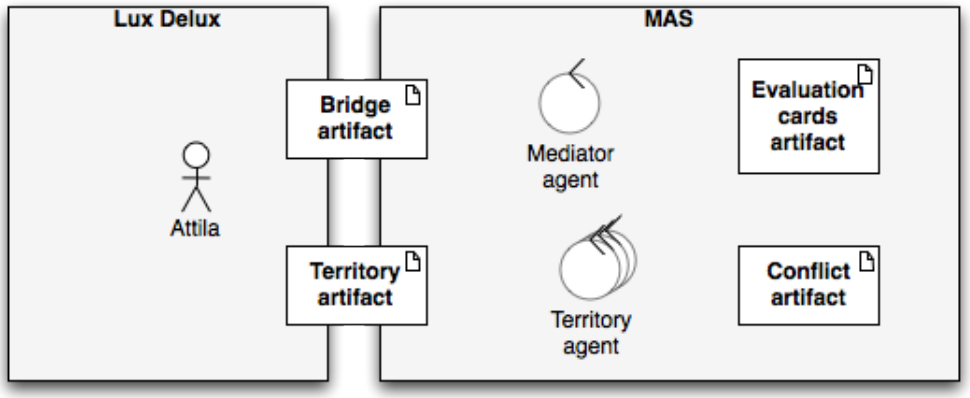

Figure 1: System architecture.

2.2 Interaction

In Attila there are two main interaction forms between MAS entities, that will be analyzed below.

2.2.1 Interaction with artifacts

Artifacts are objects without control flow, they are used by agents to sense the environment or to retrieve informations about it. An agent can interact with an artifact mainly in two ways:

- *Using operations*
  In this way the agent uses an operation on the artifact, the artifact executes this operation producing a result that can be a state change of the artifact or a return value.

- *Focus on an observable property*
  In this way the agent doesn't use the artifact directly when it needs

it, the artifact updates observable properties when a change in the environment happens. The agent perceives the change and its knowledge base is modified to be consistent with happened changes.

In Attila, both methods are adopted. Generally the first method is adopted when artifacts are used as support incorporating algorithms or functions helpful for the agents, whereas the second method is adopted when the agents must be updated due to an environment change.
The main drawback with first method is simultaneous use of artifacts from lot of agents; in this case requests are sequentialized, so there is a significant decline of performance and there is a loss of benefit of having a concurrent/distributed system. With second method the artifact notify agents and these ones will have new informations in their knowledge bases (artifacts aren't proactive, they react on environment changes to notify agents).
In Attila's MAS there are four different artifacts, each one has its role in interaction with external resources:

- Bridge artifact
  When the bot enters in a game phase, communicates this to bridge artifact that updates an observable property, the artifact blocks the bot to wait for completion of service. The mediator agent perceives the change and finds out the game phase, it can keep on elaborating the strategy and it will use the artifact to notify to the player its work completion. The player will be unblocked and it will continue its game. During each game phase, mediator agent uses bridge artifact to achieve some operations (e.g. attack, placement, movement of armies to fortify) on game platform so it can modify the environment. This is a border artifact because it makes a bridge between player and MAS.

- Conflict artifact
  This artifact works only inside MAS because it is used to access external resources to agents (algorithms, in this case) and it doesn't interface with game platform. This artifact works only with operations, so agents use it, then it returns one or more values.

- Evaluation cards artifact
  This artifact is used only by mediator agent during evaluation cards phase, it has only one operation that returns best cashable cards in a set, so even this artifact is used to access to an external resource (evaluation algorithm).

- Territory artifact
  The territory artifact interacts (where for interaction we means that the agent uses the artifact) with both mediator agent and territory agents but in different game phases. The two types of agents have in common that at the beginning of each game phase, except for cards phase, they perceive from territory artifact the actual state of game board (e.g territory name, territory value, owned territories). Other interactions with territory artifact are done in different game phases from each agent type. This is a border artifact, it is used by agents to interface directly with game platform and retrieve informations from board.
  The mediator agent uses the artifact during fortification phase to ask update about clusters[1] properties, to know shortest path between two territories through friendly territories and to know how many armies a territory can move. Last information (movable armies) is retrieved using an operation (and not an observable property) because it changes, during fortification phase, each time armies are moved into a new territory. Territory agents interact with this artifact during placement phase and use it to know if a goal path[2] has a continent bonus or to know armies of enemy neighbors of last goal path territory.

[1] A cluster is a contiguous set of owned territory.

[2] A goal path is a sequence of enemy territories, it represents a possible "path of conquest" starting from an owned country.

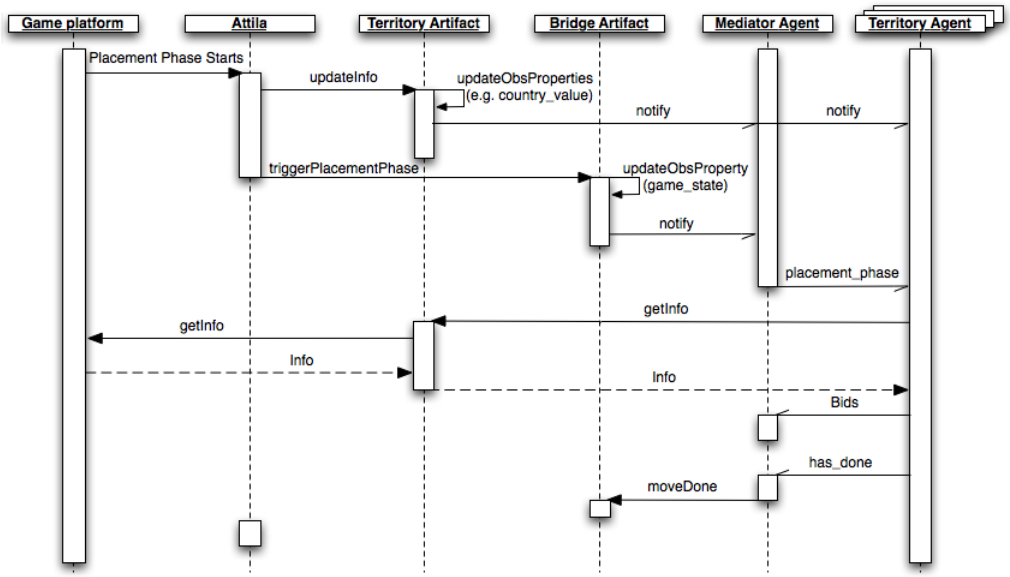

Figure 2: Sequence diagram (one iteration) during a placement phase. Method *getInfo* represents an example of information retrieving from game platform through artifact.

2.2.2 Interaction between agents

Agents, as proactive and reactive entities, need communicate between them to split jobs, to learn or cooperate to get a common goal. In our MAS, interaction between agents is used mainly to split jobs, create rendez-vous and discover artifacts inside MAS. In MAS working cycle occurs typical interactions between agents, analyzed next.

- Tell initial beliefs to an agent
  When a territory agent is created its knowledge base is empty and there is the necessity to fill it with the mediator name (receiver of territory agents values) and its associated territory (each territory agent is linked with a territory). This is done by sending a message from mediator agent to each territory agent immediately after its creation.

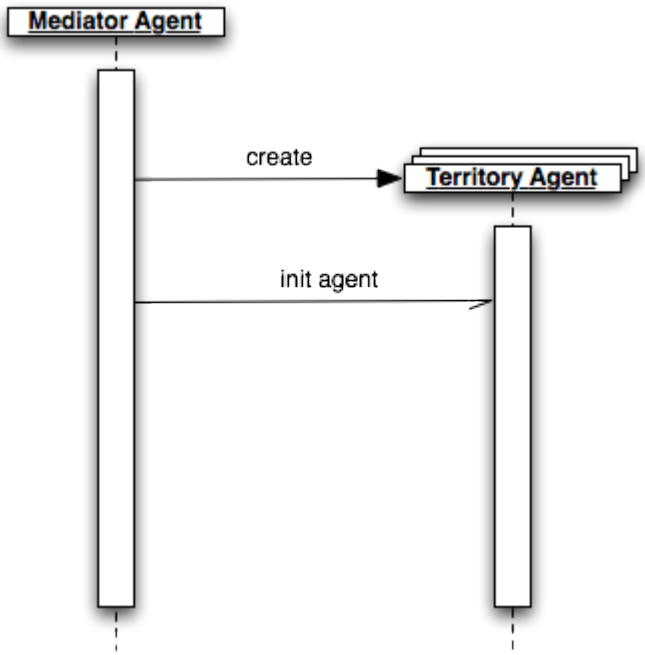

Figure 3: Sequence diagram that represents the creation of territory agents and their initialization made by mediator agent.

- Artifact names request
  During territory agents initialization, each territory agent asks to mediator agent the needed artifacts name with a message; the mediator answer the requests and puts out the name of the artifact, so each agent is able to lookup it.

- Placement phase
  The placement phase is the only one with communication between mediator agent and territory agents because it's the heaviest from the computational point of view. When the mediator agent perceives from the game that the placement phase starts, it communicates to territory agents the event, in this way each territory agent is woken up to do its work. At the end of territory agents work, each one sends obtained results to mediator agent.
  The mediator agent, to ensure that all territory agents finished their work, before processing received results creates a rendez-vous. Each

territory agent, at the end of its work, signals to mediator the completion; when everyone has finished, mediator agent is able to continue working.

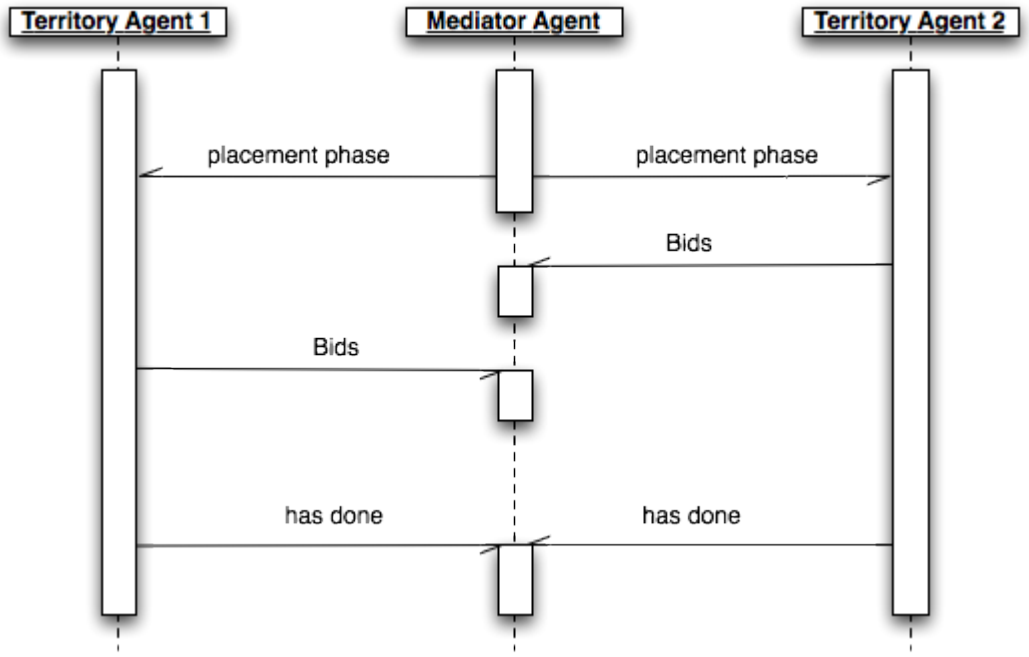

Figure 4: Sequence diagram of mediator and territory agents interaction in placement phase. Mediator agent after the notification of placement phase creates a rendez-vous and waits for territory agents work completion.

2.3 Behavior

The main actor in Attila MAS architecture is mediator agent. It's the only agent created at MAS startup, and the only one responsible on building up system architecture. Territory agents are the other entities capable of reasoning, helping mediator (also with artifacts) to take decisions about strategy actuation. With reference to system architecture and interactions depicted previously, the behavior of the two main entities of the system, relatively on the various game phases, is explained in the details.

2.3.1 System startup

Mediator agent At system startup, mediator is the only agent living in MAS; initially it has beliefs about bridge and territory artifact names, the ones that need references to the game environment and that are created by bot. By these names it's able to find out the two already created artifacts in the workspace. It also creates the two missing artifact, for cards evaluation and conflict chances, needed, respectively, in card phase and attack phase.
After the artifacts initialization phase, it starts to building up the architecture. It provides territory agents creation, with initial beliefs about the territory assigned to each one, and indirectly its reference name, to make territory agents able to communicate with itself.

Territory agents The lifecycle of territory agents begin after the initialization made by mediator. After that, each of the territory agents is created, learns its reference territory and mediator name, and starts artifact discovering phase. Differently from the mediator, every territory agent must ask to the mediator for needed artifacts; it will answer with artifact name, and than each territory is able to lookup for that artifact. Especially, the requested artifacts for territory agents are territory artifact and conflict artifact, both for elaborate values valuable for the mediator in placement phase.

2.3.2 Trade in cards

In this game phase, only the mediator agent is involved.
It wakes up when the game phase started, by observing the game state observable property of bridge artifact.
Then it gets from bridge artifact the values of cards actually owned by the player, and asks to evaluation cards artifact if the cards set is valuable to get extra armies; in that case, the artifact returns the best combination of cards.
After that, exploiting bridge artifact, possibly selected cards are traded, obtaining extra armies.

2.3.3 Place armies

Armies placement is the main part of the game where the two types of agents really cooperate to obtain the best territories where to put the best number of armies. Results of this phase are really important and draw the guidelines for next attack phase.

Mediator agent Like every game phase start, mediator wakes up and obtains by observable properties of bridge the actual state of the game, especially the number of armies available to be placed on board.
After that it proceeds to starts territory agents work, sending informations about the game phase and the number of armies to place. Then it waits for territory agents responses.
This phase its a sort of auction; every territory agent proposes an offer, where it asks the mediator for a certain number of armies to assign to a certain territory, elaborating a value associated to the territory, that represents the amount of the offer. All bids[3] elaborated by territory agents are sent to the mediator, that can now proceed by choosing the best ones.
Mediator agent retrieves all bids and sorts them by decreasing value. Then it starts, using bridge artifact operation, to assign armies to territories following informations of territory agents, until remains no armies. If there are no more bids, but some available armies, mediator assigns them to the first best bid.

Territory agents Every territory agent is contacted by the mediator, regardless it refers to an owned territory or not. If the territory is owned by the player, than the agent work starts; else, it signals to mediator the end of its job (actually, that no job was played).
Initially the agent starts to elaborate every possible goal path, starting from its territory, with a defined length (parameter of the system). To do this it exploits observable properties of territory artifact, that lets the agent know, for every territory, the list of enemy neighbors.
This elaboration produces a list of every possible path that can be followed starting from the owned territory. If the produced list is empty, it means that the territory has no enemy, so it must be an internal country and needs no more armies.
All of this paths will now be elaborated, assigning them an estimated value, with number of necessary armies, to conquer the territories of the goal path or to defend the owned territory (producing offensive and defensive bids). Then these bids are sent to the mediator agent for final auction.

[3]For bids details see subsection 4.3.3

2.3.4 Attack

Attack phase is completely realized by mediator agent, exploiting the previous placement phase results to make reasonable choices.
When game phase starts, it begins retrieving ordered bid list. In this way, attack is done starting from most valuable targets. It scans the list until remains no bids, and for each goalpath associated to bids tries to perform attacks.
After the selection of the goalpath associated to currently considered bid, it starts to attack the enemy territories in goalpath, until it conquers the entire goalpath or it looses, and can't keep on attack. To do so it uses operations on bridge artifact (to performs attacks) and conflict artifact (to evaluate attacks win chances).
The attack win chances returned by artifact are compared, for each attack, with a reference win chance; mediator performs the attack only if calculated win chance is greater or equal to reference win chance; this win chance increase after the first winning attack [4].

2.3.5 Fortify

Fortification phase ends every player turn; like attack phase, only mediator agents is involved. To implement fortification, mediator uses operations of bridge artifact.
When mediator agent wakes up at the beginning of game phase, it updates information about actual clusters owned by the player.
The concept of cluster in this phase is really important, because in each cluster, border territories became armies buyers and internal territories became armies seller. There is the need of increase cluster defense, moving internal useless armies towards border territories, attachable by enemy players.
So mediator starts the division of clusters territories in buyers (with positive number of enemy neighbors) and sellers (with positive number of moveable armies and no enemy neighbors).
The need of armies for buyers is proportional to territory value (i.e. if a territory has more enemies, or enemies with more armies, it has more value); every seller will try to sell armies to its closest buyer. If more than one buyer has the same distance, armies will be assigned proportionally to territory

[4] See subsection 4.3.1

value.
Following the *common house rule* of Risk, explained in Section 1, armies can be moved for a total of one step for each turn. So, if the closest buyer found is an adjoining territory, armies can be moved directly in buyer territory; else, the armies can only be moved in an adjoining territory, member of the path towards the buyer.

3 Tools

3.1 Lux Delux

Lux Delux [6] is a Risk game developed by Sillysoft. It is a commercial product, but its price is not prohibitive. Lux Delux gives the possibility to aspirant developers to build a personal bot in a relative easy way.
Moreover Lux comes with 11 different bots, each one with a different AI personality. They are divided into three categories (easy, medium and hard) and can be used to test the effectiveness of developer's bot (moreover they obviously are virtual adversaries for common human player to play against).
Sillysoft provides also a Sillysoft SDK, that contains everything is needed to write a personal AI for Lux Delux. It includes API documentation and the source code of all the AIs already present in the game. The programming language used is Java.

3.2 Jason

Jason [2] is a Java-based interpreter for an extended version of AgentSpeak, an agent-oriented programming language based on the BDI architecture. Jason is implemented in Java and is available Open Source, distributed under GNU LGPL. The API documentation is available on the website.
The possibility to develop BDI-oriented agents is very important in our project, especially considering the aim to extend it following the adversarial activity model [8]. Moreover it is supported by latest releases of CArtAgO (see subsection 3.3), making it easier to follow the Agent & Artifact meta-model.

3.3 CArtAgO

CArtAgO (Common ARTifact infrastructure for AGents Open environments) [1] is a general purpose framework, based on the Agent & Artifact meta-model, that makes it possible to program and execute virtual environments for multi-agent systems.
CArtAgO is implemented in Java and example on how to use it are available on the website.
Although CArtAgO is not bound to any specific agent platform, is very useful in multi-agent systems based on the BDI architecture. Moreover it directly includes a bridge for the Jason interpreter.
CArtAgO has been developed with the aim to be a simple programming model to design artifact-based environments, possibly structured in multiple workspaces (even distributed across the network ones).

4 Implementation

4.1 Bot creation and integration in Lux Delux

4.1.1 Developing and inserting a custom bot

Sillysoft, producers of game environment Lux Delux, provides also a java SDK to develop custom AIs. Developing a custom bot is quite simple. It must implement interface *LuxAgent*, that contains public methods called by Lux Delux environment during the game; implementing such methods gives opportunity to create a custom behavior for the new bot. These methods are:

- *setPrefs*: this method initializes bot, is called by game next to bot creation. It passes a copy of game Board object, so it's very important because this object allow the bot to communicate with game environment; it passes also a bot Id to identify the bot in the game.

- *pickCountry*: depending on selected game options (if the option "Selected" for territories is chosen), this methods is repeatedly called (in the initial phase of territory assignment) by game to select territories, until all have been assigned.

- *placeInitialArmies*: just like previously described method, this is called only if the relatives game option is selected ("armies placed"); the game

will repeatedly calls this method in the initial phase, after territory assignment, until remain no armies.

- *cardsPhase*: as the name suggests, this method is called at the beginning of each turn, taking as parameters the array of cards possibly owned by player.

- *placeArmies*: at every turn, after cardsPhase call, this method is called and the agent gets the amount of armies to place on board as parameter.

- *attackPhase*: after armies placement, the game calls this method to possibly attack enemy countries.

- *fortifyPhase*: at the end of turn, the method is called and armies can be moved between owned countries, to fortify borders and implements defensive strategy.

This is the main structure of custom created bots in Lux Delux environment. It's simply a Java class implementing bot strategies.
To load a new bot in Lux Delux, the resulting *.class* file must be positioned in *Support/Agents* directory inside of Lux Delux installation directory.

4.1.2 Including external libraries

Unlike the creation of simple bots, where the only requested thing to do is compiling a java class and moving files like said previously, in our case MAS technologies have to be integrated in the system, to allow the bot to exploits them in game phases to play according to its strategy. To do such a thing, it's necessary to make bot able to use MAS library, especially the ones implementing Jason and CArtAgO services, and MAS implementation, with created agents and artifacts.
So we unpacked *.jar* packages of Jason, CArtAgO and c4jason into Lux Delux installation directory, creating *jason*, *cartago* and *c4jason*[5] directories.
To import MAS implementation, we only moved *mas* build directory (with *.asl* files of agents and *.class* files of artifacts) in Lux Delux installation directory. This is the specific set of action necessary to make our system works, but it's also a general strategy to provide bots with external libraries.

[5]This *.jar* includes also some CArtAgO *.class* files, that must be joined with the ones in CArtAgO main directory.

4.2 Technologies integration

4.2.1 Launching a Jason MAS from a Java application

Jason provides an environment to develop agent based applications with a graphical interface and some tools for debugging, it's perfect for a standalone MAS. Instead, our system must be connected with an already developed platform, so the MAS has to be created directly from the bot. Other limitations are imposed by Lux Delux game framework; in fact, the game reads at startup *.class* files representing bots and instantiates them depending if they are selected for the match or not. So, it's not possible to start a bot from MAS, but there is the imperative of starting MAS from bot. Jason APIs provide *RunCentralizedMAS* class that permits the creation of a MAS from a Java application; the class reads from a *.mas2j* file the configuration and starts the system. MAS launching is included in *MasThread class*; this class launches a separated thread to start MAS, so its control flow and the one of the bot are separated.
As well as Jason is developed, it doesn't allow dynamic configuration of a MAS because it takes as input a static *.mas2j* file. To avoid this limitation Attila writes its own configuration file during the bot startup (inside *setPrefs*). Moreover, it's helpful to create mediator agent with an initial knowledge base, so in this method Attila passes to mediator artifacts names. Since the MAS must use artifacts, is necessary to specify inside *.mas2j* file the nature of agents architecture and environment; using CArtAgO requires to create a *c4jason.CartagoEnvironment* environment and agents must have *c4jason.CAgentArch* architecture.

4.2.2 Using CArtAgO to interface a MAS with a Java framework

Our bot exploits MAS to take decisions about game phases, implements its strategy and makes smart choices.
In a game match, many bots of the same type can be run together. It makes sense to think *MAS as a service*, launched by first created Attila bot, and used by every Attila bot in the game.
MAS service must be created and launched before game starts, to be available in each next game phase. Following the bot structure from Lux SDK, MAS can be created at bot startup, in *setPrefs* method, called just after bot creation. So, an Attila bot checks if a CArtAgO node is already running, by

method *CartagoService.isNodeActive*[6]; if it's not running, the service must be created, starting a *CArtAgO node* and creating a new context; then, is necessary to create artifacts that need reference to game Board, particularly bridge and territory artifact, the ones that interface game environment with MAS. Every Attila bot that comes after, and checks that CArtAgO node is already active, only lookups for the already created artifacts.
As said previously, bridge and territory artifact work like a bridge between game and MAS; especially, bridge plays the role of coordinator, notifying MAS that a particular game phase has started. Moreover bridge artifact itself must be notified of the particular game phase; Attila bot has the "thread of control" of game evolution, because Lux Delux environment calls each time its public methods relative to the game phase. So, in each of this methods, bridge operations (and in case territory artifact operations, to update observable properties) must be invoked; this is done using *doAction* on the CArtAgO context previously created. This method take as input the operation to invoke and possibly a list of parameters.
By this way, bridge artifact learns the game phase and starts MAS reasoning cycle, updating observable properties that trigger agents work. It's also necessary to guarantee synchronization between MAS and game environment; game match cannot continue until MAS work is done. This is done creating a *@GUARD operation* in bridge artifact, that blocks bridge operations (and so, game control flow) until agents signals the end of operations, unblocking the guard.

4.3 Implementation details

4.3.1 Observable properties

Observable properties are perceived from agents like environment changes, the system uses this method both to signal events that happen in game platform and to pass informations to MAS through artifacts. The two situations are analyzed below.

Observable properties to signal events At the beginning of each phase the observable property *game_state* of bridge artifact is updated, first argument is numeric and suggests the current phase (e.g. *game_state(2)* represents

[6]Available from CArtAgO 2.0.2.

placement phase).
Mediator agent has some plans that reacts to a *game_state* belief change, each plan reacts depending to first argument value. When a plan is completed, using the operation *moveDone* on the bridge artifact, first argument is changed to *10* (corresponding to move done) and the guard is satisfied. This is the complete cycle of a phase change.

Observable properties to pass informations Observable properties to pass informations are used about in all game phases; depending on the artifact that creates and updates them, here is a list of the observable properties:

- Bridge Artifact:
  - *armies_to_bid*
    It represents the armies a bot can place during placement phase, the argument is the number of armies.
  - *win_percentage*
    It represents a constraint during attack phase, because it governs bot's attack. This percentage is updated during conflicts; in first attack, reference win percentage is 0.25; after the first winning attack, its value is updated to 0.7375. This update permits to try the first conquest to get a card, being more careful in following attacks.
- Territory Artifact:
  - *player_id*
    Since MAS is conceived as a service, each player that uses it has to identify itself and update this property. The player identifier is used to retrieve informations about owned territories on the game board. The argument is numeric and is given by the game platform.
  - *country_enemy_neighbors*
    It contains the list of enemy neighbors of a territory; the territory is the first argument, the second argument is the list of enemy neighbors.
  - *country_value*
    It represents the estimated value of a territory; first argument is

the evaluated territory, second argument is the evaluation. For detailed evaluation refer to subsection 4.3.3.

- *country_is_mine*
  It indicates if a territory is owned by the player; the first argument is the territory, the second is a boolean: true if owned, false if not owned.
- *country_off_armies*
  It represents offensive armies of a territory; offensive armies are total number of armies except one (from Risk rules [7]). First argument is the territory, second argument is the number of offensive armies.
- *country_armies*
  It represents the amount of armies of a territory; first argument is the territory code, the second argument is the number of armies.
- *country_name*
  It represents the name of a territory, corresponding to a certain code. First argument is the territory code, second argument is a string indicating the name.
- *number_of_countries*
  It represents the number of countries of the board; the game can be started with different maps so the MAS has to be generic and adaptive to the contest. It has only one argument that is numeric.
- *cluster*
  It represents a cluster, the argument is the list of countries that composes the cluster. Each time an agent asks for an update to the artifact, all cluster properties are erased and defined again because their cardinality can change significantly during the game.

Each *country_** property varies its cardinality accordingly to the number of board's territories. On standard Risk board, each one of these properties has cardinality 42.

In earlier implementation, Attila was based on the retrieve of informations with operations on artifacts: this caused a loss of performance due to the sequentialization of used operation. Then we turned these operation into the perception of observable properties that guarantee faster retrieve of information. Evaluation of territory bids is still the heaviest part from the

computational point of view, that makes Attila's reasoning cycle slower than others bot, but this is caused mainly by the recursive part of the calculation (see subsection 6.1).

4.3.2 Agents communication

Agent communication occurs between mediator and territory agents in different moments during system execution.
Communication exploits basic mechanism of jason and CArtAgO.
For asynchronous message passing *.send* primitive it's used, in the two version of *tell* (that sends to target agent a message, creating a belief in its belief base) and *achieve* (that triggers a specific plan in target agent).
For creating synchronization between agents, it's exploited the *blackboard* artifact, created by default in CArtAgO environment, that can be used with tuple spaces semantic, with *in* and *out* primitives to retrieves and putting tuples in a shared tuple space between agents.
These methods have been used both individually and together, in different phases of system implementation.

System startup After MAS creation, mediator agent discovers the already created artifacts and creates the others in *setup* plan. After that it continues to build up system architecture, creating territory agents, executing *initialize_architecture* plan. Here it creates as many agents as the number of territories of the map, and informs them on the territory each one has to manage. To do such a thing, it uses a *.send(Agent name,achieve,init_agent(Territory code))*. Through *achieve* mode, the plan *init_agent* is called on *Agent name* territory agent; in this plan, each territory agent creates its beliefs on mediator agent name and country code, and can start to retrieve its necessary artifacts. Mediator agent name is necessary for territory agents to send it messages for various purpose, as described below.

Territory agents artifact discovering Mediator agent is the only one that knows the names of artifacts available in MAS, so territory agents have to request him the names of the artifacts they need.
When each territory agent is created, it only knows the types of needed artifacts, and learns mediator name taking it in the source field of *init_agent* plan, called by mediator by an achieve *.send*. To discover real artifact names, it calls *findArtifact* plan, with a parameter representative of the needed

artifact (bridge, territory, eval_cards or conflict). In this plan, it uses a *.send(Mediator name,tell,request_info(Artifact))*. Trough *tell* mode, mediator agent receives a new *request_info(Artifact)* belief in its belief base. It reacts when this belief appears, and retrieves from its belief base the requested artifact name. At the same time, the territory agent that sends the request, is blocked on the execution of an *in(Artifact,Artifact name)*. The mediator makes an *out(Artifact,Artifact name)* that unblocks the territory agent with the answer to its request.

Placement phase This game phase is the heaviest from the computational point of view, and involves both mediator and territory agents.
When mediator agents wakes up after the change of the *game_state* observable property, it starts territory agents work calling, for each territory agent, *.send(Agent name,achieve,placement_phase(AvailableArmies))*. In this way it makes the territory agents start their *placement_phase* plan, passing them the number of available armies that have to be placed on board. Then, it waits for agent work completion, blocking itself with an *in(has_done)*. Each territory agent, if referred to an owned territory, starts its work, calculating bids; else, it executes an *out(has_done)*, signaling that its work, in this case nothing, is done. The remaining working agents, when bids are ready to be sent to mediator, execute a *.send(Mediator name,tell,bid value)* for each bid to send, then make an *out(has_done)* too to signal work completion.
When mediator agent receives all of the 42 *has_done* tuples, it can proceed with the rest of game phase management.

4.3.3 Bids calculation

Placement phase is the most difficult and also important for game progress. Choices made at this moment influence the subsequent phases, especially attack.
Like introduced in subsection 2.3.3, placement phase is a sort of auction: every territory agent of an owned territory sends its bids to get a certain number of armies. A bid is a tuple (actually it's a belief in mediator belief base, see subsection 4.3.2) that contains:

- *territory code*: numeric value gave by game framework to each territory (from 0 to 41), useful to distinguish between various territory bids.

- *bid value*: numeric value estimating how much a goal path, starting from a territory, is valuable to have.
- *goal path*: chains of territories associated to bid, it represent a possible path from that territory to enemies territories, with a specific length (in our implementation, goal path length is 5 countries, bigger values causes a great loss of performances).
- *armies to bid*: best number of armies to get the goal path.

It's necessary to distinguish between two different types of bids: there are *offensive bids*, estimating how much a goal path is valuable, depending also on probability to conquer it, and there are *defensive bids*, that measures the need of extra armies to defend a certain territory, depending on its value and its neighborhood situation. The two types of bids will now be analyzed.

Offensive bids Offensive bids are numeric estimations of a goal path importance. To do such estimations, is necessary to consider different variables:

- goal path value (w): this value it's simply the sum of all territory values in goal path, in addition with a continent bonus, that increase the value of the goal path if it permits to complete a continent, and get the relative armies bonus.
- probability to conquer goal path (p): it's the chance to finally get the goal path territories from the home territory; in this calculation, is supposed that the territory gets the entire amount of armies for placement phase, and this value is compared with total number of enemy armies. The probability is obtained trough Conflict artifact, also used for attack results predictions.
- probability to defend goal path at the end (d): it's the chance to defend the possibly conquered goal path at the end of battle; it is based on the estimation of the remaining number of armies when the goal path is finally conquered.

The different goal paths of a territory are analyzed, using this values to compare them and estimate what is the best goal path. The formula used to elaborate final goal path value is:

$$bid_{off} = \frac{p_i w_i d_i}{i} \quad i = 1, \ldots, N \tag{1}$$

In this equation, N represents the total number of armies available in a placement phase; in order to estimate what is the best number of armies to ask to mediator agent, the best goal path value is recalculated for each number of armies from 1 to N (a previous selection of the best goal path is done supposing that territory gets the entire amounts of armies).
Each bid value is on a per army basis. This makes them easy to compare.

Defensive bids Defensive bids measures the need of armies of a territory, respect to its number of armies and armies of enemy neighbors. The goal is to improve the ability to defend itself from external attacks, considering all of enemy neighbors armies. The formula for bids calculation is the following:

$$bid_{def} = \frac{v * d_{(i+m)}}{i} \quad i = 1, \ldots, N \tag{2}$$

The value of defensive bid is given from two values: the territory value v and the defense probability d, calculated with current number of armies plus the number of armies to bid for.
Territory value is composed of two parts. First part is fixed, and depends on the position of the territory on the board (i.e. number of borders, portion of the continent occupied and continent bonus).
The second part is variable and related to the particular situation of game match (e.g. how much of the territory's continent we own, how many friendly neighbors has, how many armies do the enemy neighbors and friendly neighbors have). For more details on territory value, see [4].

Changes respect to original implementation Referring to [4], the strategy to calculate offensive bids in particular was partially changed, both from heuristic and performance point of view.
From heuristic point of view, differences from original implementation are essentially two:

- the value of a goal path is the sum of all territories and not only the value of the last territory.
- continent bonus in our implementation is not assigned only if a territory is the last to conquer to own a whole continent, but is assigned even if a path (so the conquest of a list of subsequent territories) allows to conquer the whole continent, to promote bids that guides towards this achievement.

From performance point of view, there are important changes, starting from placement phase in general and then also in offensive bids calculation.
In the original implementation, the auction for armies bid is repeated, until no more armies are available. This means that, at every cycle, the best bid is chosen, the relative number of armies is placed, and then auction starts again, getting another best bid. This was not acceptable from computational point of view: often the territory with the first best bid, got also the second, and so on. Given the structure of MAS, with high number of communications between agents and artifacts, this led to long computation time.
To improve performances, the algorithm was partially changed. In final implementation, there is only an auction; here every agents involved sends to mediator four bids, two defensive and two offensive; in mediator, these bids are sorted and armies are assigned, until remains no armies. If no more bids are available, but yet there are some armies to place, the remaining armies are assigned to first best bid.
The second great difference is in offensive bids calculation. In the original implementation, for every goal path, the number of available armies was iterated, to get the best number of armies for each goal path. In final implementation, the best goal path is chosen supposing that it gets all of the available armies. Then, only for the best goal path, the number of available armies is iterated to get the best amount to reach this goal.

4.3.4 Dice odds

Odds calculation takes an important role in Risk: the result of every conflict[7] is determined by rolling of a proper number of dice. Knowing the odds of winning an entire conflict (or, better, a sequence of conflicts) is necessary in order to calculate the bid for a goal path.
Starting from known odds (see table 1 and table 2)[4], we developed a recursive algorithm that, given the number of armies for the attacker and for the defender, creates a table, useful to calculate the odds for the attacker and the defender to win the conflict. For example, for a five vs four conflict we build a 6x5 table. Each cell represent the odds to have the number of remaining armies equals to that cell's coordinates at a certain point of the conflict: in previous example, at coordinates [5,4] (five and four are starting numbers of

[7]Battle: a single dice roll (maximum values are three dice for the attacker and two dice for the defender; each die represents an army). Conflict: a sequence of battles that ends with the victory of one of the two contenders.

armies) there must be the value 1, because this is obviously a certain event. Using values of table 1 and table 2 the algorithm recursively fills the table with correct odds. This results in having a row representing zero remaining armies for the defender and a column representing zero remaining armies for the attacker: summing this values will give respectively the odds of the attacker and the odds of the defender to win the conflict.
The multi-agent system needs to retrieve odds values for a conflict very often. In order to improve performance by several orders of magnitude, we create a unique table (using the previously described algorithm) with already calculated odds for several values of offensive and defensive armies. If the number of offensive and/or defensive armies exceeds the table dimensions, we scale them down and approximate the conflict with a smaller one with the same proportions between armies.
The artifact that provides this service is the conflict artifact. It also provides an advanced operation that allows the agent to calculate the odds to accomplish an entire goal (composed by more territories), estimate the number of remaining armies and the odds to defend the last territory of the goal (useful in bid calculation).

Table 1: Dice odds for battles with one defending die

	1 attacker 1 defender	2 attackers 1 defender	3 attackers 1 defender
Attacker wins	15/36 (42%)	125/216 (58%)	855/1296 (66%)
Defender wins	21/36 (58%)	91/216 (42%)	441/1296 (34%)

Table 2: Dice odds for battles with two defending dice

	1 attacker 2 defenders	2 attackers 2 defenders	3 attackers 2 defenders
Attacker wins	55/216 (25%)	295/1296 (23%)	2890/7776 (37%)
Defender wins	161/216 (75%)	581/1296 (45%)	2275/7776 (29%)
Lose one each		420/1296 (32%)	2611/7776 (34%)

5 Experiments

Risk is intrinsically a not deterministic game, so it's interesting to test Attila in some different situations, calculating his winning percentage to evaluate his effectiveness.
We used the same map, the classical one, and the same rules set for every match: random initial territories, placed initial armies, "5-5-5" configuration for exchanging cards (see subsection 1.2). What change is the enemies set. Attila faced three different categories of bots, following the Lux Delux classification: easy, medium and hard (see [4] for a short description of the used bots). A fourth category was defined, including random enemies. This last type of match should be interesting because it shows the capability of the bots to adapt to several different strategies.
The parameter used to evaluate the effectiveness of the bots is the winning percentage, that measure the capability to win a match. Average ranking is an interesting parameter too, but it has to be interpreted taking into account the strategies of the bots involved: some bots have strategies that make them really hard to defeat in the early turns of the match, but without being more effective in winning a match.
For each of the four categories, we ran a relative low number of games (more than 40, however), so the statistics are quite indicative, even if it's easy to identify a clear trend for the effectiveness of Attila.

5.1 Attila versus easy bots

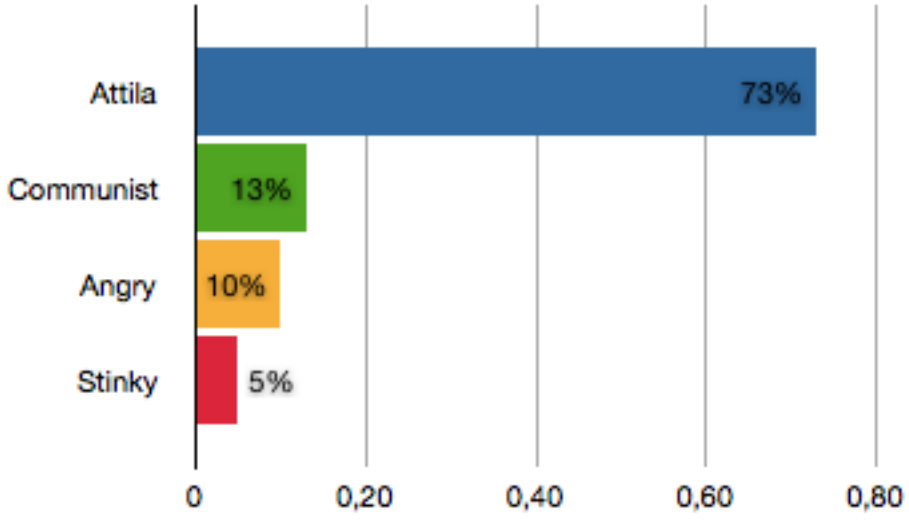

Figure 5: Winning percentage in matches involving Attila and the easy bots.

In matches against easy bots, Attila has the highest winning percentage (73%): it is clearly the most effective bots among these. This also reflects on Attila to have the best average ranking (1.65): the result is very impressive, due to randomness of Risk game. It shows a good defense ability in earlier turns of the game and a very good effectiveness in exploiting the many weaknesses of easy bots.

5.2 Attila versus medium bots

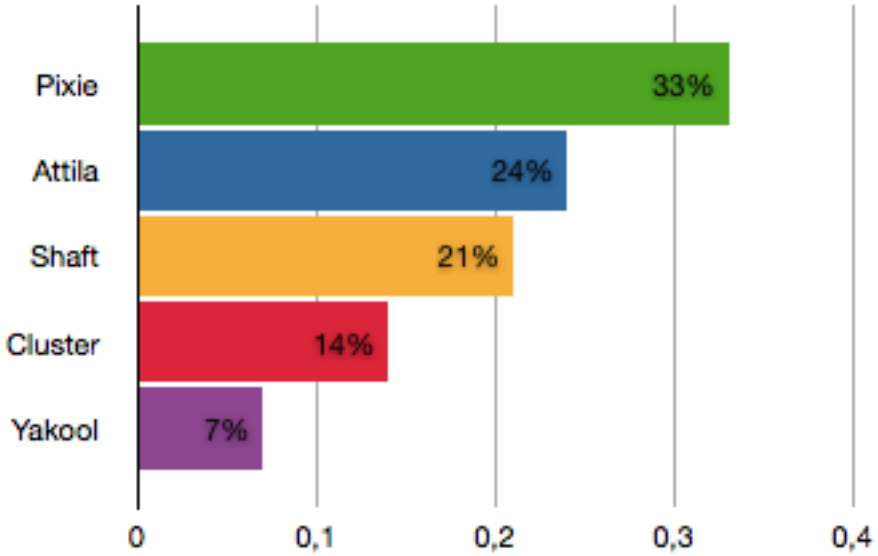

Figure 6: Winning percentage in matches involving Attila and the medium bots.

Statistics of matches against medium bots are quite good:, Attila placed second with 24% of victories, not so far from the first (Pixie, 33%) and with performance comparable with the third (Shaft, 21%). It placed third in average ranking (3.24), a little less of a medium ranking (3.00).
This set of matches shows a first weakness of Attila: it has a very offensive strategy that lead it to be very weak in first turns of a match. Some bots (all except Shaft) take advantage of this weakness and this results in Attila to have a high chance to be defeated quickly. Otherwise when Attila succeeds in conquering a significant part of the world map, his chance to win the match significantly increase.

5.3 Attila versus hard bots

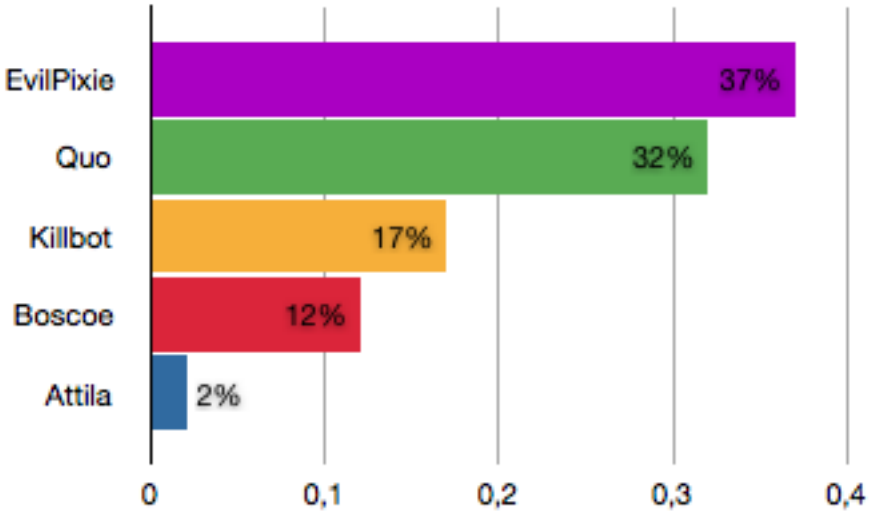

Figure 7: Winning percentage in matches involving Attila and the hard bots.

Attila ranked last against hard opponents with a very low effectiveness (only 2%, corresponding to a single victory in more than 40 matches played). Attila ranked last also in average ranking (4.22), far from the medium value (3.00). Hard bots are merciless against careless bots (like Attila): it's very easy for them to exploit the weakness of a player not very good in defending his territories.
This is the worst result achieved by Attila: with improvements proposed in subsection 6.1 we want to improve the general effectiveness of our bot, in order to increase this ranking.

5.4 Attila versus random bots

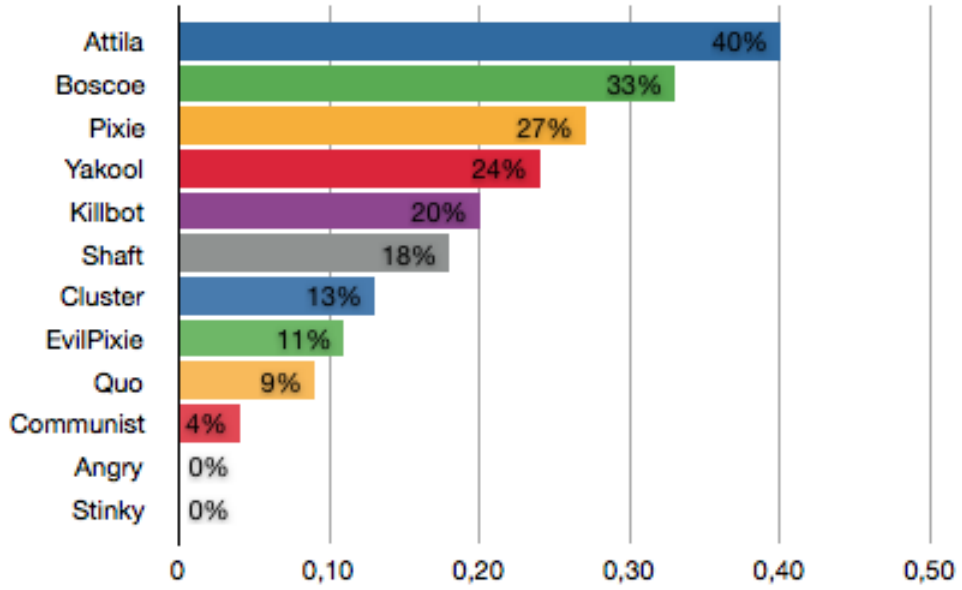

Figure 8: Winning percentage in matches involving Attila and random bots.

This category of match is different from the previous one: the set of opponent is not fixed, but is randomly determined choosing five from the eleven available bots. In order to assure that every bot plays the same number of matches than the others, for every match run with six random bots, a second match with the remaining six bots has run.
Attila ranked first with 40% of victories. This is quite unexpected, because the result is even higher than the one achieved against medium bots. The average ranking (3.11) is not far from the medium value (3.50) and comparable with the results achieved by others bots (most of them have a similar value).
A possible explanation of this result is that if in the enemies set there are bots with low defending abilities (like easy bots, against which the effectiveness of Attila is demonstrated), our bot is one of the best in exploiting their weakness, succeeding in conquering a significant part of the world map. When this happens, the effectiveness of Attila seems to be comparable with the effectiveness of hard bots.

6 Future works

6.1 Possible extensions

Attila was intended to be quite neutral in the definition of the strategy: most of his actions are based only on the evaluation function (see 4.3.3). There are many possible extensions based on our Risk experience that could exploit the agent architecture in order to significantly improve Attila effectiveness.

- Territories could cooperate to conquer another territory (i.e. attacking it one after the other).
- If two territory agents bid for the same goal path, it could be better (eventually) to not divide assigned armies between the two territories.
- Now the mediator agent doesn't care if a territory needs to be defended. The territory agent could communicate to the mediator that has a defensive bid as best bid, in order to inhibit the attacks from that territory.
- In the fortification phase, only internal territories are possible seller, while also border territories should eventually give some armies to a needy neighbor.
- The odds to defend a goal is now intended as the odds to defend the last territory of the goal path. It is possible to have an improvement by considering it as the odds to defend all the territories that are part of the path.
- Sometimes an owned territory "steals" an enemy territory to a neighbor which has received armies in placement phase for a path including that enemy territory: the agents should be able to recognize other alternative paths (if they exist).
- Path deep in recursive algorithm could be dynamically related to the "breadth of the tree", in order to not waste too much time (paying the cost to have shorter paths in some situations).
- The recursive algorithm is now implemented in the territory agents: it could be implemented in an artifact in order to improve performance.

6.2 The adversarial activity model

The adversarial activity model [8] is a formal BDI based model for rational agents. This model defines the mental states (beliefs, desires and intentions) of agents living in an environment where the entities involved have conflicting interests that should lead to adversarial (and not cooperative) interaction. The model is defined for zero-sum environments, where players gain or lose exactly what opponents lose or gain, so it's suitable for Risk game.
The model is described in term of axioms, intended to be a guideline to implement an adversarial activity based agent. The idea is to extend the BDI architecture of Attila (currently strongly based on a heuristic function) implementing behaviors that follow the adversarial activity principles.

References

[1] CArtAgO. `http://cartago.sourceforge.net/`. Last visited on 2011-05-03.

[2] Jason. `http://jason.sourceforge.net/Jason/Jason.html`. Last visited on 2011-05-03.

[3] Stefan J. Johansson and Fredrik Olsson. Using multi-agent system technologies in risk bots. In *Proceedings of the second artificial intelligence and interactive digital entertainment conference*, pages 42–47, 2006.

[4] Fredrik Olsson. A multi-agent system for playing the board game risk. Master's thesis in software engineering, Blekinge Institute of Technology, 2005.

[5] Andrea Omicini, Alessandro Ricci, and Mirko Viroli. Artifacts in the A&A meta-model for multi-agent systems. *Autonomous agents and multi-agent systems*, 17:432–456, 2008.

[6] Sillysoft. Lux Delux. `http://sillysoft.net/lux/`. Last visited on 2011-05-03.

[7] Wikipedia. Risk. `http://en.wikipedia.org/wiki/Risk_(game)`. Last visited on 2011-05-03.

[8] Inon Zuckerman, Sarit Kraus, and Jeffrey S. Rosenschein. The adversarial activity model for bounded rational agents. *Journal of autonomous agents and multi-agent systems*, 2010.