BDI Agents as Players in Real-time Strategy Games

Andrea Dallatana

Abstract

The aim of the project is to work on an already existing RTS game and exploit agents to develop the artificial intelligence of the units.
The chosen target is 0ad : an open source game already in its final stages of development that uses high end graphics and refined mechanics to deliver a simulation of historical warfare and economics.
The first step was to understand how the game's AI was accessed and developed: at the very core of 0ad there is a powerful open source game engine, called "Pyrogenesis", that manages various aspects of the game, including graphics, menus and player interaction; all of the game logic, including the AI, is instead developed using javascript and xml, then interpreted at runtime by the engine using Mozilla's SpiderMonkey.
The work then consisted in the design and development of a multi-agent system platform, to support the control of the game's entities using BDI agents logic. Since the platform had to be in javascript, to be integrated into the game, it was developed from the ground up albeit it followed loosely Jason's reasoning cycle and AgentSpeak's language constructs.
The last part was to develop some basic behaviours for the agents controlling units and buildings, to test the validity of the created platform and of MAS in the current contest.

Contents

1 Analysis of the target infrastructure

1.1 0 A.D.

0 A.D. is a free, open-source, cross-platform real-time strategy (RTS) game of ancient warfare.
In short, it is a historically-based war/economy game that allows players to relive or rewrite the history of Western civilizations, focusing on the years between 500 B.C. and 500 A.D. The project is highly ambitious, involving state-of-the-art 3D graphics, detailed artwork, sound, and a flexible and powerful custom-built game engine.
The goal of the game is to accumulate resources to build armies and structures and then eliminate the enemy players while defending from their attacks.

1.2 Pyrogenesys and spidermonkey

The game engine, written in C++, provides the underlying structure for the whole game, managing graphics, sound and user inputs.
All gameplay mechanics are instead written in javascript and interpreted by the engine using Mozillas Spidermonkey.
Unit basic behaviors like stances, moving into formation, pathfinding are all part of the game logic written in javascript and then interpreted.
This also provides a set of objects containing the current state of the game with the condition of all entities and all relevant data used by the game logic.
The game state is updated every 150ms, actually dividing the game into turns; its the engine that gives visual continuity to these turns by displaying the correct movement animations, allowing the graphics to appear fluid.

1.3 Game logic and AIs

The game logic provides the underlying structure to support a set of higher commands by the player: it establishes a set of possible commands and then gives them a determined result into the game.
For example the command Walk to x,z is executed by analyzing the map and possible obstacles for the unit that wants to move and then a pathfinding algorithm finds the best course.

All possible commands are provided directly using the Engine.PostCommand() function directly or by using the appropriate APIs like the Entity objects.

```
selectedEntity.move(x,z)
```

Game logic is common between all players, humans and AIs, the difference stands in the fact that these commands are executed by the game interface in case of a human player, by the AI scripts in case of a computer controlled player.
A common AI API gives the basic function to interact with the game entities, but a Gamestate object is also created containing all entity arrays and relevant data and also a way to quickly access them.

1.4 Current AIs

All the current AIs function like centralized algorithms that evaluate the current game state and then try to actuate a set of plans adapting to it.
They are omniscient, meaning they know enemy and resources positions everywhere on the map, even through fog of war and outside units vision range.
The main AI, QBot, works by doing a cyclical analysis of the game conditions and actuates a plan to better its own and to support its other plans; it also adapts to enemy movements and strength by trying to have always a better and stronger army.
It builds a strong economy able to support a constant flux of soldiers and provides a strong challenge, although having a unmatchable knowledge of all that is on the map.
All units and structures usually receive orders in batches matching a selected plan of action.

2 Analysis and planning of the agent based architecture

2.1 Objectives

The objective of an agent based AI would be to decentralize all the thinking and to provide the units and buildings independent thought.
Every unit and structure would be a BDI agent with his own reasoning cycle, collaborating with other agents to achieve immediate goals that present themselves from the evolving of the match; also trying to reach the main goal of winning the game.
Since every agent would be reactive to its environment, another objective of this AI would be to avoid cheating by being omniscient; instead trying to match the human players conditions and giving a fair challenge.

2.2 Requirements

A MAS-based AI would need an infrastructure to allow individual agents to perceive around them, to evaluate their condition, to communicate with other agents and to act on a set of plans.
This infrastructure would also need to allow quick and easy developing of plans, perceptions, messages and all agent related actions.
Another requirement would be to integrate all this into the existing AI interface provided and to make all work in the shortest calculation-time possible.

2.3 BDI Agents

Belief-Desires-Intentions agents adequately fit the profile of providing the right constructs for this type of AI.
A unit as a BDI agent would have:

- a set of beliefs that consists of the ones given at its creation, the ones acquired through its perception of the world and other agents messages and also as the result of its own reasoning;
- a set of perceptions that function like sensors inside the game world;
- a set of desires given by the nature of the agent and new ones given by other agents as the result of their reasoning;
- one or more desires that verify their actuation conditions become intentions and then execute as agents actions.
- game interaction functions that work like effectors for the intentions and actuate commands to the engine.

2.4 Platform decision choices

Once understood the inner workings of how the games engine and AIs relate to each other, using an encapsulated interpretation of logic scripts, it became clear that the best course of action would be to create the entire framework of the MAS inside the given structure.
The other choice would have been to create a bridge between an existing BDI Agents platform and the game engine.

2.4.1 Pairing an existing platform with the game engine:

Advantages:

- Access to an established protocol for defining agents;
- Wide range of languages and tools to aid the development of the agents;
- Speed and efficiency of the reasoning cycle of the agents, both thoroughly tested given the diffusion of the existing platform;

Disadvantages:

- Extremely difficult bridging between the platform and the game engine with a certain need to rewrite entire parts of the latter;
- Possible loss of efficiency in terms of speed in the communication process between the platform and the games engine;

2.4.2 Creating a new "ad hoc" platform:

Advantages:

- Flexibility in the design to adapt to the target infrastructure;
- Possibility to develop ad hoc solutions to situations in the game logic;
- Custom reasoning cycle tailored to best suit the current context;

Disadvantages:

- Need to create a complete and functional platform from the ground up;
- Not ideal context to build a MAS both in term of speed and overhead;

The choice made was to create a new platform using javascript and seamlessly integrated inside the engine like other AIs, but following an existing platform as a guideline to plan its development; Jasons platform and language (based on AgentSpeak) was selected to be adopted in that regard, as its combination of simplicity and power made it best suited for this type of context.

3 Development of ABot the Agent-based AI Bot

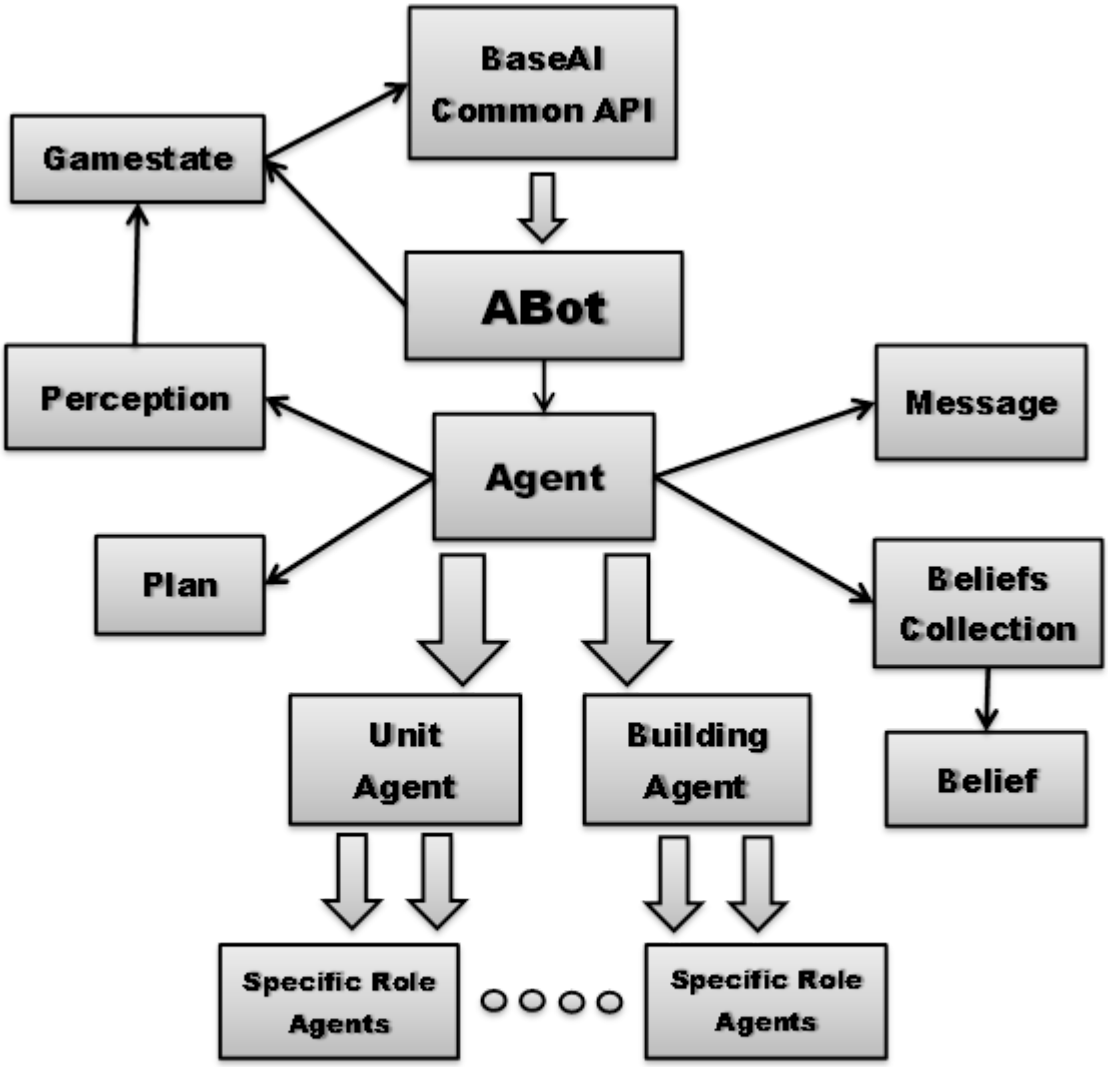
3.1 Outline

The AIs main object is ABot, extending the BaseAI interface; the onUpdate method is called every game turn and its where all the reasoning cycles methods are called.
The Agent object contains all the methods related to the MAS infrastructure and also a common configuration to all agents.

The objects representing the BDI nature of the agents are:

- BeliefsCollection and Belief, that provide a way to store, retrieve and update beliefs;
- Message, that function as a mean to encapsulate communications between agents;
- Plan, that represent both a desire, if not triggered nor matching its conditions, and an intention in case it is ready to be executed;
- Perception, that working paired with the Gamestate object provide the sensory input from the game data;

There is then a distinction between units and buildings, where both of them inherit a basic configuration matching their specific role; from there every different entity acquire in the same way the behaviors that distinguish their task.


BaseAI
Common API
Gamestate
ABot
Perception
Message
Agent
Plan
Beliefs
Collection
Unit
Agent
Building
Agent
Belief
Specific Role
Agents
Specific Role
Agents

3.2 ABot Object and the Main Cycle

The AIs root object, ABot, contains the array with all the active agents and on every gameturn repeats the main cycle that match with the agents reasoning cycle.

updateAgentsArray Searches into Gamestates entity array for the AIs own entities, then creates new agents, if they are missing from the current agents list and triggers the die method of the ones missing; each Agent is created with a unique ID that is the same as the one its associated entity has inside the game.

LookAround For every alive agent it cycles all his active perceptions.

Mailman Collects all messages from the agents outboxes, reads their delivery address, in the form of their ID, and then inserts them in the corresponding inboxes.

agentsCheckMail Every agent checks its inbox and then acts on the content of the messages.

Think This is where all triggered plans are checked to see if their conditions are met, if so they are activated as intentions.

Action All intentions are executed.


Update Agents Array
New Agents
Internal Data
Dead Agents Deletion
LookAround
Field of Vision
Status Perceptions
Mail Delivery
Mail collection from Agents'
Mail Delivery to Agents' Inboxes
Mail Check
Mail opening and context updates
Think
Triggered Plans'
Act
Intentions Execution

3.3 Beliefs

Beliefs are a collection of data that the agent believes to be true or false.

Every Belief consists of 4 attributes:

- Name
- Key
- Data
- BelieveTrue

Every tuple of Name and Key represents only one Belief, while there may be more Beliefs with the same name.
Data is a hashtable containing all the desired fields; a Data entry could be associated with an array of variables.
BelieveTrue is used to support strong negation; if a Belief has this attribute set to false then the agent believes it to be false and that is different from being unknown.

The syntax to create a new Belief is:

```
var data = {    Attribute  : field ,      };
var belief = new Belief(Name, Key, Data, BelieveTrue);
```

Example

```
Var data  =
        {   PositionX  : 234,   PositionZ  : 453,
          SurroundingUnits  :  [3452, 4563, 1345] };
Var belief1 =
    new Belief( VisibleEnemies  , 1345, data, true);
```

Every agent has his own BeliefsCollection which contains all his beliefs: this object provides the methods to access the beliefbase to add new beliefs, get the existing ones or delete them.
If a Belief with the same (Name, Key) as an already present one is added, then it triggers an update, which checks the differences between the two Data items and adds the new information.

There are two optional methods, onUpdate and onPerception, that are meant to be code to be executed in case the Belief receives an update or for its initial perception by the agent.

3.4 Perceptions

Each agent has a collection of sensors that let it perceive the state of the game around it and these are defined as Perception objects.
Perceptions are created with an unique Name, a reference to the agent using them and an attribute Continuous; then the Perceive method has to be implemented if we want the Perception to do anything.

The Perceive function usually consists of:

- access to Gamestate to retrieve the relevant data;
- creation of a new Belief or a collection of them using said data;
- addition of the Beliefs created to the beliefbase of the agent;

Not all the Perceptions are active all the time, in fact they only function if they are explicitly activated; once a Perception is active it will work in the next LookAround cycle, but to become inactive again after that.
Only if a Perception has the Continuous attribute set to true it will remain active for each turn; then the only way to deactivate it will be to call the deactivatePerception method.

Synthax:

```
var perception =
   new Perception("Perception Name",
                  this, continuous true/false);
perception.perceive =  function (gamestate)
{
        perception body
};
this.addPerception(perception);
```

Example

```
var perception =
     new Perception("RegisterMyPosition", this, true);
perception.perceive =  function (gamestate)
{
        var data =
                { "x": this.agent.position[0],
                        "z": this.agent.position[1] };
        var belief =
                new Belief("KnownPosition", "me",
                        data , true);
        this.agent.addBelief(belief);
        belief =
                new Belief("KnownPosition", this.agent.id,
                        data , true);
        this.agent.addBelief(belief);
};
```

3.5 Plans

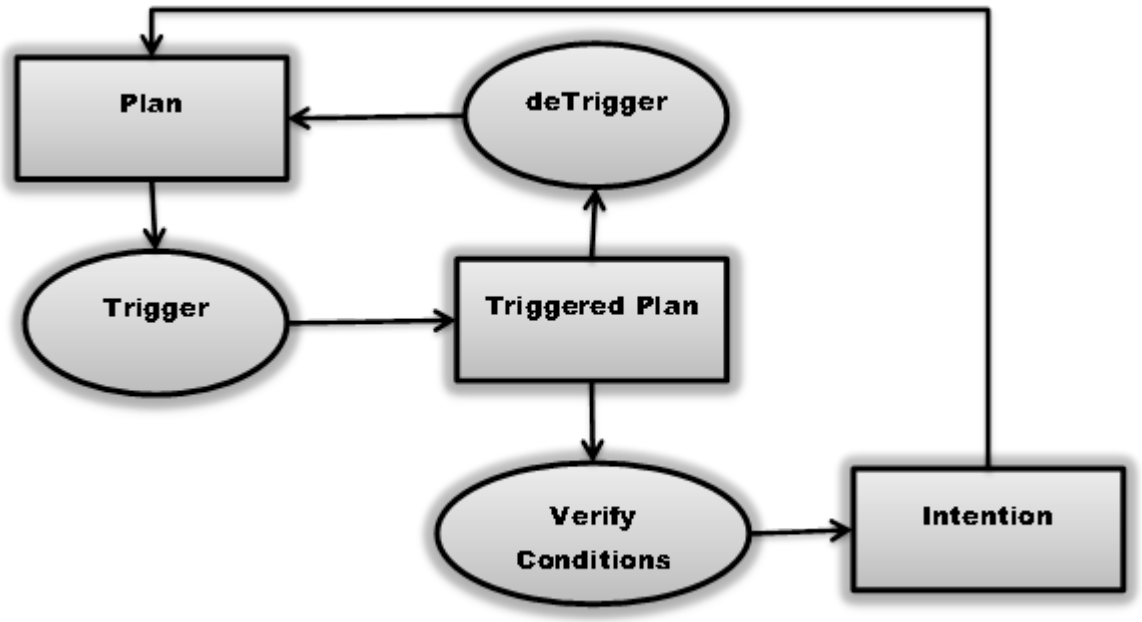
A plan represents a desire that could become an intention and its the means with which the agents interact with the world around them.
Each Plan is unique by Name and the only other attribute is Sender, which is used if the Plan has been sent by another agent.
There are three methods:

- Execute, that is the code that will run when the Plan becomes an intention;
- verifyConditions, that is the check to be made in each Think cycle to see if the Plan will become an intention, if not implemented it will always return true;
- onTrigger, (optional) that is like a subplan, something that will be executed when the Plan is triggered and is usually used to help the agent meet the Plans conditions.

Each Plan once created and then inserted into the agents planbase has to pass two steps to become an intention: first someone has to trigger it and then it will have to pass the conditions check, only triggered Plans are checked.

The execute method of a Plan can contain:

- Creation, access or deletion of Beliefs;
- Creation of messages to send to other agents;
- Creation, trigger, detrigger, deletion of Plans;
- Creation, deletion, activation, deactivation of Perceptions;
- Interaction with the games world by using the appropriate functions for each agent, like WalkTo, Construct, Repair, Garrison, ;



Synthax:

```
var plan = new Plan("Plan Name");
plan.execute =  function (agent)
        {
                Plan body
        };

plan.verifyConditions = function (agent)
        {
                Conditions
                return true/false;
        };
        Plan.onTrigger  = function (agent) {
                subplan
        };
this.addPlan(plan);
```

Example

```
var plan = new Plan("Survival");
plan.execute = function (agent)
{
    agent.triggerPlan("Escape");
    agent.triggerPlan("Survival");
};
plan.verifyConditions = function (agent)
{
    var enemyCountArray =
          agent.getBelief("VisibleEnemyUnitCount");
    var enemies = 0;
    if(enemyCountArray.length > 0){
          for(var type in enemyCountArray)
          {
              if(enemyCountArray[type].key == "champion")
                    enemies += enemyCountArray[type].data["count"];
          }
    }
    var ownCountArray = agent.getBelief("VisibleOwnUnitCount");
    var own = 0;
    if(ownCountArray.length > 0){
          for(var type in ownCountArray)
          {
              if(ownCountArray[type].key == "champion")
                    own += ownCountArray[type].data["count"];
          }
    }
    if(enemies > (own + 2)) return true;
    return false;
};
this.addPlan(plan);
this.triggerPlan("Survival");
```

3.6 Messages

Agents can communicate with others by using Messages: these, once created and sent, will be deposited in the next turn in the inbox of the respective receivers.
A Message has four attributes:

- Sender, containing the ID of the agent sending the Message;
- Receiver, containing the ID of the agent to which the Message will be sent, this is an array and can contain multiple IDs;
- Type, defining the role of the Message and what it can contain;
- Content, hashtable containing different kinds of data depending of the Messages Type.

Type	Description	Content
Achieve	Triggers a Plan in the receivers planbase.	PlanName
Abandon	Detriggers a Plan in the receivers triggeredPlans list.	PlanName
Tellhow	Sends a new Plan to be added to the receivers planbase	Plan
Forgethow	Makes the receiver delete a Plan inside its planbase.	PlanName
Tell	Adds a Belief to the receivers beliefsCollection or updates an existing one.	Beliefs (Array of Beliefs) or BeliefName, BeliefKey, BeliefData (Object) to trigger a data update
Forget	Removes a Belief from the receivers beliefsCollection or a Data entry from one of the existing ones.	BeliefName, BeliefKey (optional), BeliefData (Object) (optional and requires BeliefKey)
Ask	Makes the receiver send a Tell message to the sender, containing the Belief asked.	BeliefName, BeliefKey (optional), BeliefDataFieldName (optional)

Each type of message has his own method inside the Agent object; there is also another version for each of them called Broadcast, that will send the same message to all known agents.

Example

```
this.sendForgetBroadcast("KnownAgents", this.id);
```

This will make all other agents forget about the sender and it is used in case of an agents death.

4 Agents and Behaviors

4.1 About the Agent Object and Modularity

The Agent object contains all the methods that compose the MAS infrastructure, but also some basic Beliefs, Perceptions and Plans, shared by all agents.
Every agent has an initial configuration that changes based on his role and this is composed by all the modules it inherits.
For example an infantry soldier would have a different set of behaviors than a fortress: the first inheriting from Agent UnitAgent SoldierAgent WorkerAgent and the other from Agent BuildingAgent TrainBuildingAgent DefenceBuildingAgent . .
This kind of modularity permits to categorize determined roles and then attributing every agent the corresponding one based on its nature.

4.2 Locality vs Omniscience

The first consequence of using an agent based solution to the development of the AI is the possibility to implement locality.
Every agent has a Perception called LookFOV (Look Field of View) that analyzes the Gamestate looking for all entities inside its range of vision and then registers them as a set of Beliefs.
Centralized AIs usually implement an omniscient knowledge of the world, all enemies and resources are known and so is their position on the map; this clearly gives them an advantage but also transmits a feeling of artificial to the human opponent.
An omniscient AI knows the position of the enemy armies, their strength and composition, where instead the locality AI only knows what it can see, like a human player.
The advantage of deploying locality in an agent based AI is how quick and easy becomes the development of responsive behaviors: a simple Survival plan, that makes the agent consider how many enemies and allies it sees around it and then decides if the battle is a lost cause, is a big step towards both efficiency and realism.

4.3 Performances and Benchmarking

Using the profiling system provided by the engine it is possible to see how many milliseconds each step of Abots reasoning cycle takes.
There are still too few behaviors developed to make a complete picture, but using the basic perceptions and test plans it is possible to see that the system uses very little resources, with the exception of locality perceptions.
The current APIs are developed to be used by centralized AIs so the process of looking through all the entities to see if they are in the agents range of vision and catalogue them into beliefs takes a considerable amount of time.
QBot, the main centralized AI, let its algorithms run only one time every five turns to reduce its impact on performance; this approach could be taken by ABot distributing each cycles step on a different turn.

QBot
Average Cycle Time: 30ms

ABot
Average Cycle Time with locality: 55ms
Average Cycle Time without locality: 5ms

4.4 Conclusions and future development

BDI Agents are suitable for AI development in Real Time Strategy because they allow easy implementation of complex behaviors with a low impact on performances.
The only downside found was the overhead given by the locality perceptions but it could be better assigned to the lack of optimization of the APIs than to the weight of the MAS.
Now that the framework is in place future developments could be directed to advancement in agents behaviors and optimizations both in terms of speed and efficiency.

References

[1] 0 a.d. , http://wildfiregames.com/0ad/

[2] 0 a.d. development wiki , http://trac.wildfiregames.com/wiki

[3] Mozilla Spidermonkey , https://developer.mozilla.org/en/SpiderMonkey/

[4] Pyrogenesis Engine , http://www.moddb.com/engines/pyrogenesis

[5] Jason , http://jason.sourceforge.net/Jason/Jason.html