Argus - A BDI Multi-Agent Environment in Minecraft

Bardia Parmoun
Carleton University
Ottawa, Canada
bardiaparmoun@cmail.carleton.ca

ABSTRACT

This document provides a detailed explanation of a BDI Multi-Agent development environment based on the game Minecraft. In this environment, the BDI agents serve as non-playable characters (NPCs) in a complex and competitive scenario. Using this environment, this paper aims to demonstrate the capabilities of the BDI paradigm for implementing core multi-agent principles using the `Jason` language. In addition, two sample strategies along with their performance are thoroughly discussed. It is ultimately concluded that Jason and BDI are very suitable for multi-agent use cases.

KEYWORDS

BDI, Jason, Multi-Agent Systems, Minecraft

1 INTRODUCTION

1.1 Motivation

Autonomous Agents are often described as computer systems that follow a constant cycle of observation and reaction to their environment to achieve an overall objective [8]. A popular paradigm when designing such agents is the belief-desire-intention (BDI) model, which aims to describe the behaviour of agents through the lens of practical human reasoning. BDI groups the different aspects of the agent into the three concepts of beliefs, desires, and intentions [5]. Such abstractions help make the agent's reasoning process more transparent and explainable; however, finding the correct level of abstraction and properly defining the agent in the BDI can be challenging for complex real-life examples.

AgentSpeak is a formal language for describing BDI systems. This language describes the behaviour of the agent through a set of small procedural plans that are only activated through certain conditions and achieve a specific set of goals [4]. The Java implementation of AgentSpeak, Jason, is notable due to its extensive library of internal actions, as well as its support for single- and multi-agent environments [1].

The applications for AgentSpeak, Jason, and BDI are typically straightforward and theoretical. Most applications of Jason do not account for multi-agent environments, despite the language having extensive support for them. This paper aims to assess the capabilities of Jason by providing a complex Jason-compliant multi-agent environment. Through this environment, it is ultimately shown that Jason, and ultimately BDI, are quite suitable for implementing complex, multi-agent strategies. Furthermore, two distinct strategies for this environment (aggressive and conservative), with varying sizes of coalitions, are introduced and thoroughly analyzed as starting examples, with the hope that more complex strategies will be developed in the future. From this initial analysis, it is concluded that being aggressive and forming big coalitions are key factors to being successful in this environment!

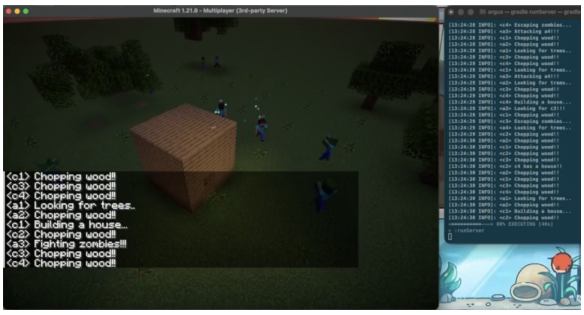
1.2 Problem Description

The environment utilized for this paper is based on the game Minecraft, which is a popular game developed by the Mojang studio. The main objective of the game is for the players to explore the various worlds in the game while defending themselves against the enemies. The main gameplay surrounds the idea of players mining different elements from the world and building various objects with them. It also has full extensive support for multiplayer and modifications, which has led to it becoming very popular over the recent years [11].

The world chosen for this study, Argus, is a custom modification of the Minecraft flat world, consisting of a set number of zombies and trees. In Argus, the players are modeled as autonomous BDI agents with the main objective to survive the zombie attacks while collecting the highest number of points. In this model, each game lasts for 1.5 minutes, and it will be cut off after that. As part of their gameplay, the players are free to attack the zombies and other players; chop trees and collect wood; build houses to hide and recover; and form alliances with other players to share resources. Figure 1 illustrates a demo of Argus taken from:

https://youtu.be/j8UhS9gPfME?si=hS4u_QGpgfPuyVkP.

Figure 1: A demo of Argus.



2 RELATED WORKS

There have been very few attempts at integrating BDI and Multi-Agent systems. A notable example is [6], which explores the use of `distributed Multi-Agent Reasoning System (dMARS)`, another implementation of AgentSpeak, in a simple MAS problem. This approach focuses on the lack of learning in the BDI paradigm and utilizes Logical Decision Trees to figure out when the agent's plans are ready. This approach also allows for multiple agents in parallel. This paper focuses a lot on extending BDI architecture to allow individual agents to learn their planning behaviour; however, it does not focus on the interactions between these agents, which is of interest for multi-agent research.

Another example is [9], which explores a really simple simulation of a food delivery system. In this simulation, the various stakeholders, such as platforms, riders, shops, and customers, are all modeled as BDI agents. This paper explores two different algorithms for the intention selection process, which is a crucial step in BDI. The two considered approaches are `Monte Carlo Tree Search (MCTS)` and `Insertion Heuristics`. The paper then concludes that MCTS is far better performance-wise than insertion heuristics. This is a very interesting finding and could be used to improve any BDI=MAS application. Unfortunately, Jason does not use either method when it comes to intention selection.

BDI has also been applied to a MAS context from a semantic web lens in this project [2] that aims to make BDI agents that interact with a semantic web network. The project utilizes the domain-specific modeling language `Semantic Web Enabled Multi-Agent Systems Modeling Language (SEA_ML)` to create a simple semantic web network as a small multi-agent environment for a group of BDI agents to interact with. This environment is an electronic bartering (E-barter) system where customer agents match and barter items using ontologies and the `semantic web service (SWS)`. Due to BDI's reactive nature and its support for intention stacking, the project was a lot more successful in matchmaking the barters. This is a significant finding for multi-agent research, as it demonstrates that BDI can be beneficial in expressing agent preferences.

It is worth noting that Jason has been considered as an alternative or even an improvement in most of the above-mentioned papers; however, an example of a project that resembles this work the closest would be the usage of Jason in the `Multi-Agent Programming Contest (MAPC)` [7]. In this challenge, the BDI agents compete in the well-known Gold Miners scenario, which is a grid-like environment where the agents need to navigate the obstacles in order to find and collect the gold. This paper highlights many important aspects of MAS, such as the agents' ability to communicate with one another and share beliefs to distribute tasks and react to dynamic changes in the grid world. It also highlights the BDI architecture's ability to continually update plans based on new perceptions, allowing agents to constantly re-plan in a dynamic environment. Finally, this paper shows that the BDI paradigm has clear support for role-specific agents. Many ideas from this paper were adopted for the two sample strategies that were created for Argus, with the motivation to see if they will still hold up in an environment more complex than that of the Gold Miners.

Finally, an important paper for Minecraft development is [10], which is a gym-like Minecraft platform to evaluate the performance of different multi-model multi-agent models. The project essentially consists of a large dataset of 55000+ task-variant images and support for various autonomous players using `Large Language Model (LLM)` agents. Some of the tasks mentioned in this project, such as building, attacking, and farming, inspired the scope of Argus. This project illustrates the vast features of Minecraft and the complexities of the decisions that a single player can make. It also demonstrates that although the large language models seem capable of handling a large list of actions, they can get easily overwhelmed when presented with too many novel goals or an unseen number of agents. This is an area where BDI can outperform such models with its support for dynamic intention handling.

3 BACKGROUND

3.1 AgentSpeak and Jason

The belief-desire-intention (BDI) model aims to represent the behaviour of an agent in terms of the three concepts of beliefs, desires, and intentions. At a given time, a BDI agent has specific knowledge (beliefs). It often obtains these beliefs by perceiving the environment around it; using these beliefs, the agent will follow a set of plans (intentions) that together help it achieve a goal (desire). In a typical BDI reasoning cycle, the agent perceives its environment, updates its belief base, and then selects the most applicable plan as its intention to eventually reach its desire [5]. AgentSpeak is a theoretical language that is used for specifying these plans [4]. Various interpreters have been developed for AgentSpeak. The interpreter that has been chosen for this paper is Jason [1].

Jason has a prolog-like syntax called `ASL`. In the ASL syntax, beliefs are represented as predicates. ASL represents desires in the form of achievement and test goals. Achievement goals, prefixed with `!`, represent the desire to reach a specific predicate. Similarly, test goals, prefixed with `?`, represent the desire to check if a specific predicate is true. Jason also adds a notion of triggers, which represent the addition (prefixed with `+`) or removal (prefixed with `-`) of specific beliefs, achievement goals, or test goals [1]. Using these concepts, ASL plans are then defined as:

```
trigger : context <- body.
```

In other words, a plan is essentially how the agent reacts to a specific trigger under a specific context. In addition to plans, Jason also defines rules, which are predicates that represent a group of other predicates all compounded. Rules are represented as:

```
rule :- predicate1 & predicate2 & ... .
```

Here is a simple ASL program that utilizes all of these concepts:

```
woodsChopped(20).
buildRequirement(house, 12).
buildRequirement(sword, 10).
!loop.
hasEnoughWoodFor(Woods, Object) :-
    buildRequirement(Object,OBJECT_WOOD_REQUIREMENT) &
    Woods >= OBJECT_WOOD_REQUIREMENT.
+!loop: woodsChopped(Woods) & not(hasWeapon(_)) &
↪  hasEnoughWoodFor(Woods, sword) <-
    say("Building a sword...");
    build(sword);
    !loop.
+!loop: woodsChopped(Woods) & not(houseCount(_)) &
↪  hasEnoughWoodFor(Woods, house) <-
    say("Building a house...");
    build(house);
    !loop.
```

In this program, the agent starts with beliefs about its current inventory of wood and the amount of wood it needs to build a house and a sword. It then starts with the desire `loop`, which is used as the default intention to maintain a constant cycle. The agent utilizes the rule `hasEnoughWoodFor` to figure out it can afford to build an object. In this case, it realizes that it has enough wood to build a sword and it proceeds to build one since there is no belief of type `hasWeapon` indicating it already has a weapon.

3.2 Paper, Citizens & CommandHelper

Various tools exist for creating custom Minecraft modifications. The main tool used for this project is `Paper` [12]. Paper is a Minecraft game server based on an older open-source project named `Spigot` which has extensive API support for performing custom modifications to the game. Paper's APIs are fully accessible through `Java`. Once the Java `jar` file for the server is ready, it can simply be run on a server (or as `localhost`) and accessed through the main game. In addition, Paper supports a library of plugins all in the `jar` format that can be used to extend the capabilities of the server.

A notable plugin used for this project is the `Citizens` plugin [3], which primarily deals with the creation and maintenance of non-playable characters (NPCs) in the game. With this API, various aspects of the NPCs, such as their position, appearance, and actions can be controlled. The NPCs can also be made to respond to various events in the game, such as being attacked or interacted with.

Another plugin used for this project is `CommandHelper` [13], which allows for creating custom scripts to automatically run Minecraft commands using a scripting language called `CHScript`. CHScript has a similar syntax to `JavaScript` and has full support for most programming constructs. The CommandHelper plugin was proven to crucial for automating the setup process of the custom Minecraft world used for Argus.

4 MODEL

4.1 Agents

The non-playable characters (NPCs) in Argus are modelled as the Agents. As such, the set of agents can be defined as:

$$N = \{1, 2, ..., n\}$$

In this model, the specific strategies for each agent (the AgentSpeak files), are considered roles. Currently, there are two distinct roles defined for the agents:

$$\rho = \{\text{attacker, capitalist}\}$$

The agents also have specific types for their coalition strategies. The agents can either be alone (`loner`), form coalitions with everyone (`group`), or only form coalitions with a specific type of agents (`odd` or `even`). Thus, the set of coalition strategies can be defined as:

$$\theta = \{\text{loner, odd, even, group}\}$$

Thus, each agent $i \in N$ can be defined as a tuple of its role and coalition strategy:

$$(\rho_i, \theta_i)$$

4.2 Environment

The environment is dynamic and partially observable to the agents. The environment includes the world layout; zombie positions; placements of trees and their status; and position and status of other agents. Set E includes the combination of all these factors. At a given time t, the environment can be defined as:

$$e^t = (\text{world, zombies, trees, agents}) \rightarrow e^t \in E$$

4.3 Beliefs

Set B denotes all the possible beliefs in the world, including perceptions and state information. B_i denotes agent i's beliefs.

4.3.1 Perceptions. These include what the agent can sense from its environment. In BDI, these are expressed as beliefs. Table 1 lists the perceptions for each agent.

Table 1: List of perceptions for each agent.

Perception	Meaning
`type(_)`	agent type
`woodsChopped(_)`	number of woods chopped
`buildRequirement(house, _)`	number of woods needed for a house
`buildRequirement(sword, _)`	number of woods needed for a sword
`buildRequirement(axe, _)`	number of woods needed for an axe
`buildRequirement(trident, _)`	number of woods needed for a trident
`zombieDefenceLimit(_)`	number of zombies to attack at once
`allPlayers([_, . . . , _])`	list of current players
`near(tree)`	agent is near a tree
`near(zombie, _)`	agent is near zombies (count)
`near(player, [_, . . . , _])`	agent is near a player (list of players)
`houseCount(_)`	number of houses available to agent
`hiding`	agent is hiding in a house
`health(_)`	agent's current health between 0 and 1
`hasWeapon(_)`	agent has a weapon (weapon type)
`damagedBy(_)`	agent took damage from the given entity

4.3.2 State Information. These include the information that the agent maintains internally. In BDI, these are also expressed as beliefs. Table 2 lists the state information for each agent.

Table 2: The state information for each agent.

State Information	Meaning
`lowHealthThreshold(_)`	agent considers this health level critically low
`searchTimeout(_)`	how long the agent should conduct a search
`buildRequirement(donation, _)`	agent's donation limit
`ally(_)`	agent is an ally with the given agent

4.4 Actions

Set A denotes all the possible actions in the world. A_i denotes agent i's actions. Table 3 summarizes the actions available to the agents.

Table 3: The actions available in the world.

Actions	Meaning
`say(_)`	send a message to the logs
`find(_)`	find and navigate to an entity (tree, zombie, [player])
`chop_wood`	chop a tree (if near one)
`escape`	jump to a random location to escape danger
`attack(_)`	attack an entity (zombie, [player]) if nearby
`build(_)`	build an object (house, sword, axe, and trident)
`enter_house`	go to a house (if any available)
`leave_house`	leave the house (if hiding in one)
`donate_wood(_)`	donate a certain amount of wood to the given agent
`receive(_, _)`	receive (wood, count) or (house, from)

4.5 Messages

As part of the multi-agent design, the agents are allowed to send messages to each other. BDI has full support for this and these messages allow agents to share beliefs. Set M denotes all the possible messages in the world. M_i denotes agent i's messages. Table 4 summarizes the messages available to the agents.

Table 4: The messages available to the agents.

Message	Type	Meaning
`wantAlliance(_)`	`askIf`	agent wants an alliance
`allianceConfirmation(_)`	`tell`	agent formed an alliance
`endAlliance(_)`	`tell`	agent ends an alliance
`hasHouse(_)`	`tell`	agent announces they have a house
`need(_)`	`askIf`	the asking agent needs something
`donated(_, _)`	`tell`	agent donated some entities to another

4.6 Policies

Policies include the rules for the game. These rules allow the agents to interact with the environment. The set π includes all the policies in the world. Table 5 summarizes the policies for Argus.

Table 5: The policies for Argus.

Policy	Definition
P1	Agent can find an entity (tree, players, zombies) within 50 blocks.
P2	Agent can chop a tree 5 blocks.
P3	Agents can attack zombies and players within 5 blocks.
P4	An agent can hide in a house to hide from zombies and recover health.
P5	Only two agents can hide in a house at once.
P6	Agent can share woods with its allies if they are within 5 blocks.
P7	Agent without weapon can attack 1 entity at once and escape 10 blocks
P8	Agent with sword can attack 2 entities at once and escape 8 blocks
P9	Agent with axe can attack 3 entity at once and escape 6 blocks
P10	Agent with trident can attack 4 entity at once and escape 4 blocks
P11	Swords cost 10 woods to build
P12	Houses cost 12 woods to build
P13	Axes cost 15 woods to build
P14	Tridents cost 20 woods to build
P15	A game runs for 90 seconds

4.7 Desires

In BDI, these indicate the goals of the agent. Set D denotes all the possible desires in the world. D_i denotes agent i's desires. Table 6 summarizes the desires available to the agents.

Table 6: The desires available to the agents.

Desire	Meaning
`!broadcast(_, _)`	send a message (type, content) to all players
`!sendToGroup(_, _, _)`	send a message (type, content) to the given group
`!loop`	the agent's main logic loop

4.8 Strategies

In BDI, strategies can be defined as a series of plans that the agent follows based on the corresponding events. A strategy can be formally defined as:

$$\sigma_i : (\rho_i, \theta_i, B_i, D_i, M_{-i}) \rightarrow (A_i, M_i)$$

σ_i essentially defines how an agent of a given role and type can react to a specific set of beliefs, desires, and messages. By generalizing this, the strategy space of the agent S_i can be defined:

$$\sigma_i \in S_i$$

This strategy space is ultimately what gets converted to AgentSpeak in the corresponding `.asl` files for each agent.

4.9 Intentions

In the BDI paradigm, intentions represent the plans that the agent has committed to achieving. In this model, the intentions are essentially a subset of the agent's strategy space. Thus, the intentions for agent i can be defined as:

$$I_i \subseteq S_i$$

The process of selecting the intentions from the strategy space is dependant on the implementation. Intention selection for agent i at time t can be formally defined as:

$$\iota_i^t = \sigma_i(\rho_i, \theta_i, B_i^t, D_i^t, M_{-i}^t) \rightarrow \iota_i^t \in I_i$$

4.10 Mechanism

To define a Mechanism, a formal transfer function is required to map the current world state and the agents' intentions to a new world state with a set of rewards $r_i \in R^n$. The transfer function be formally defined as:

$$g : (I_1 \times I_2 \times ... \times I_n) \times E \rightarrow E \times \mathbb{R}^n$$

So, at a given time t, for a set of intentions $(\iota_1^t, \iota_2^t, ..., \iota_n^t)$, g is:

$$g(\iota_1^t, \iota_2^t, ..., \iota_n^t, e^t) = (e^{t+1}, r_1^t, r_2^t, ..., r_n^t)$$

Putting it all together, a Mechanism $\mathcal{M}$ can be formally defined as:

$$\mathcal{M} = (N, E, I_1, I_2, ..., I_n, g, \pi)$$

4.11 Utilities

As previously mentioned, the main objective of the agents in Argus is to survive the zombie attacks while collecting the highest number of points. Formally, the utility for each agent i at a given time t, given the transfer rule, is defined as:

$$u_i : (I_1 \times I_2 \times ... \times I_n) \times E \rightarrow \mathbb{R}$$

So, at a given time t, for a set of intentions $(\iota_1^t, \iota_2^t, ..., \iota_n^t)$, u_i^t is:

$$u_i^t(\iota_1^t, \iota_2^t, ..., \iota_n^t, e^t) = r_i^t$$

Table 7 summarizes the reward structure for Argus.

Table 7: The reward structure for Argus.

Environment Update	Reward
Agent built a house	500
Agent attacked a zombie	25
Agent donated wood to an ally	50
Agent survived the game	$\frac{10000}{count(survivors)}$

Using this, the Nash equilibrium for the mechanism can be defined as:

$$u_i^t(\iota_i^{*^t}, \iota_{-i}^{*^t}, e_t) \geq u_i^t(\iota_i^t, \iota_{-i}^{*^t}, e_t) \quad \forall i, \forall t, \forall \iota_i, \iota_i \neq \iota_i^*$$

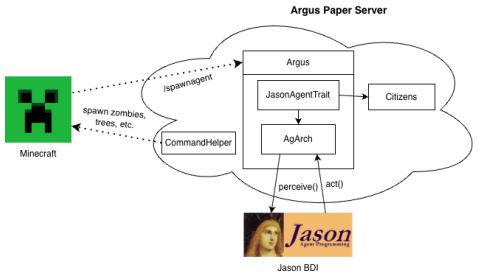
The possible existence of such equilibrium state for Argus can be shown through simulations, discussed in the Methodology section.

5 METHODOLOGY

5.1 Environment Architecture

Figure 2, shows the overall architecture of Argus for defining the agents and the environment. As previously described, the main aspect of the environment is a Paper server that Minecraft can talk to. Within this server, there are three plugins: Citizens, CommandHelper, and the custom Argus plugin. Within that plugin, a custom trait file `JasonAgentTrain` was created in compliance with Citizens. The trait describes the NPC. Finally, that trait file connects to a custom Jason agent architecture file (`AgArch`) which is Jason's way of connecting to an external environment. Using this file, Jason is able to perceive the environment and perform external actions.

Figure 2: An overall architecture of Argus.



5.2 Model Implementation

5.2.1 Agents. Each agent role is simply defined in a separate AgentSpeak(.asl) file which contains its strategy space. An agent having a particular role corresponds to what .asl file it loads. Additionally, the agent type is simply passed to it as the belief `type(_)`.

5.2.2 Beliefs. The beliefs described in the model, can be directly translated to Jason. As described in section 4.3, the beliefs consist of perceptions and state information. The perceptions are directly added to Jason through the `perceive()` function in the AgArch file. The state information is added as initial beliefs in the corresponding .asl files for each agent. Current initial state beliefs are:

```
lowHealthThreshold(0.25).
searchTimeout(1000).
buildRequirement(donation,1).
```

The implementation can be extended to support more beliefs.

5.2.3 Actions. Similar to beliefs, the action predicates are added directly to the .asl file. They are then implemented in Java in the AgArch file. This can be extended to support more actions.

5.2.4 Messages. Jason natively has a `.send(AgName, MsgType, MsgContent)` function that is used to send a message of a given type (`askIf` for questions and `tell` for statements) to a specific agent (`AgName`) with the contents (`MsgContent`). The messages then create new events for the agent. Specific beliefs can be then made from those event. It is crucial to define these message handling plans before every other plan in the file so they are the first to get triggered. This avoids the agent from missing messages.

For example, to form an alliance, a message of `wantAlliance` with type askIf is sent to another agent. Then that agent can choose to form an alliance by replying with an `allianceConfirmation` message of type tell. The agents can just choose to add an `ally(_)` belief with the other agent's name if an alliance has been formed. Here is a code snippet for it:

```
+message(askIf, Sender, wantAlliance(Sender,
↪  SenderType)): type(AgType) & not(type(loner)) &
↪  SenderType == AgType <-
    say("Received an alliance request from ", Sender);
    +ally(Sender);
    .my_name(AgName);
    .send(Sender, tell, allianceConfirmation(AgName)).
+message(tell, Sender, allianceConfirmation(Sender)) <-
    say("Formed an alliance with ", Sender);
    +ally(Sender).
```

The implementation can be extended to support more messages.

5.2.5 Policies. These are essentially the rules of game. They are all implemented in the AgArch file as part of the belief and action implementations. For example, when an agent tries to chop a tree, the code checks to see if the agent is near a tree. If an action fails a policy, a failure is sent as its status to Jason. The failure handling is then done in the .asl file as part of the policy. Similarly, beliefs also need to comply with policies. For example, if it is detected that there are two agents inside a house, that house is not counted towards the `houseCount(_)` belief for the agents that share it

5.2.6 Desires. Desires are also directly implemented in the AgentSpeak file as achievement goals. Whenever an agent wants to achieve a goal, it simply adds the goal to its desire set. Jason will then try to find a plan that can achieve that goal and add it to its intention set.

5.2.7 Strategies. In AgentSpeak, these translate to the plans for agents. A plan is defined as reacting to a particular event given a set of preconditions. In a plan, the agent can send messages or perform actions. When multiple plans are applicable, Jason will choose the first one that is applicable; so the order of plans indicate their priority. For example, here are plans for attacking zombies:

```
+!loop: health(Health) & ((damagedBy(zombie) &
↪  not(needsRecovery(Health))) |
(near(zombie,NumZombies) & canSurviveZombies(Health,
↪  NumZombies))) <-
    +attempted_attack;
    say("Fighting zombies!!!");
    attack(zombie);
    -attempted_attack;
    !loop.
+!loop: near(zombie,NumZombies) | damagedBy(zombie) <-
    +attempted_escape;
    say("Escaping zombies...");
    escape;
    -attempted_escape;
    !loop.
```

Here it is shown that for the same default intention `!loop`, the agent has two separate plans. The first plan is to attack the zombies for which the agent needs to verify that it was either attacked by a zombie or it is close to a manageable number of zombies. Only in that case, the agent attempts to attack the zombies. If neither one of those conditions are true and the agent is still close to a zombie, it will just attempt to escape. Notice how by properly ordering the plans, it is only required to list the conditions needed to satisfy a given plan without having to specify other cases.

5.2.8 Intentions. An intention is the plan that is selected by Jason for execution. Jason selects a given plan and commits to it as its intention until one of the following happens: the plan results in an action failure or a higher priority plan is selected. The process of intention selection is quite simple for Jason and it involves simply choosing the first applicable plan. Jason then maintains a stack of plans that are executing. Whenever a higher priority plan is presented, it will simply pause the current intention and switch to the higher priority intention. Once that intention is done, Jason will go back to executing the older intentions [1].

5.2.9 Utilities. The scores for each agent are tracked in their respective AgArch file. After each successful action that yields a reward, the score is updated accordingly. The scores are then reported to the plugin that manages the game and maintains all the agents. This is where that final survival reward is also calculated.

5.3 Mechanism Implementation

5.3.1 Agents. There are 8 agents in Argus.

5.3.2 Environment. The selected world for this environment is the "flat world" in Minecraft. The map is made of a 50 block by 50 block land filled with tree saplings. In Minecraft, these saplings turn into trees overtime. Additionally, 8 zombies are randomly placed on the map. The players are all loaded on the map at $(0, 0, 0)$.

5.3.3 Roles. Two main sample roles are proposed for the agents: `attacker` and `capitalist`. Table 8 summarizes these two roles.

Table 8: The different agent roles in Argus.

	Attacker	Capitalist
House Building	only build one house	build many houses
Weapon Building	build a sword, axe, and a trident	only build a sword
Building Order	sword, house, axe, trident	house, sword, house, ...
Alliances	form alliances if allowed	form alliance if allowed

5.3.4 Types. As previously mentioned, each agent is given a type based on its alliance preferences. Table 9 summarizes the types.

Table 9: The different agent types in Argus.

Type	Definition
`loner`	does not form alliances
`odd`	forms alliance with the odd agents of the same role.
`even`	forms alliance with the even agents of the same role.
`group`	forms alliance with all agents of the same role.

5.4 Experiments

To compare the effects of the roles and types in Argus, three distinct experiments were performed as follows:

- 4 (`attacker`, `loner`), 4 (`capitalist`, `loner`), and 8 zombies
- 2 (`attacker`, `odd`), 2 (`attacker`, `even`), 2 (`capitalist`, `odd`), 2 (`capitalist`, `even`), and 8 zombies
- 4 (`attacker`, `group`), 4 (`capitalist`, `group`), and 8 zombies

Each experiment was run 30 times to ensure robustness.

5.5 Repository

For the detailed implementation of this project please refer to: https://github.com/bardia-p/argus/tree/master.

6 RESULTS

6.1 Qualitative Results

6.1.1 Software Design. As shown in the methodology section, every aspect of the model was easily mapped to Jason concepts. With Jason, the design and implementation of the agents is quite transparent. Additionally, Jason provides a layer of abstraction between the agent and the environment definitions. To add new strategies for Argus, one would simply need to create more .asl files.

6.1.2 Performance. Jason was able to behave as expected in this fast-paced environment. The reasoning cycle for Jason was set to be every ticks which is just 0.2 seconds. Additionally, as explained before the Argus environment is not observable and quite dynamic; however, it was still possible to explain it via simple belief predicates. This simplification made the environment much more manageable.

6.2 Quantitative Results

6.2.1 Experiment 1 - 4 (`attacker`, `loner`), 4 (`capitalist`, `loner`). Figure 3 illustrates the results from this experiment.

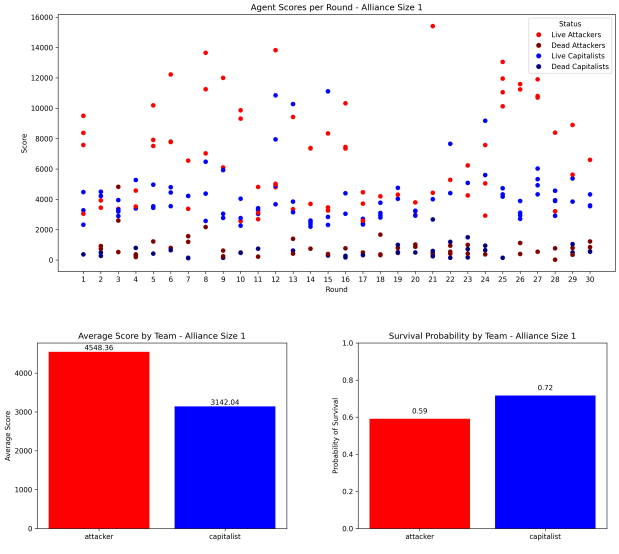


Figure 3: Individual agent scores, average score per role, and average survival rate per role for alliances of size 1.

6.2.2 Experiment 2 - 2 (`attacker`*, odd), 2 (*`attacker`*, even), 2 (*`capitalist`*, odd), 2 (*`capitalist`*, even).* Figure 4 illustrates the results from this experiment.

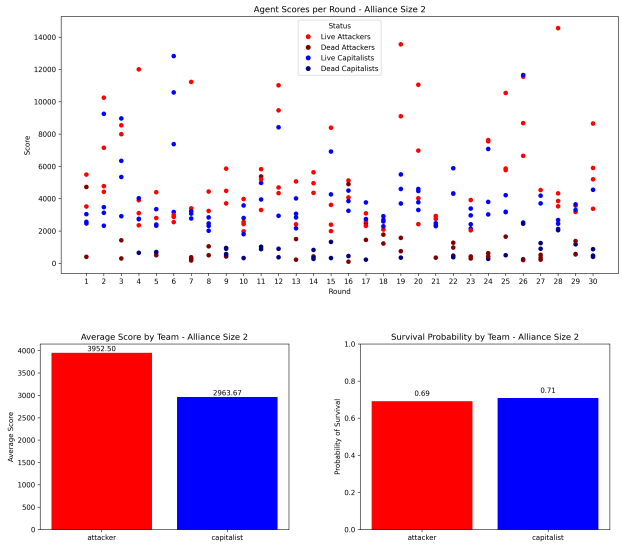


Figure 4: Individual agent scores, average score per role, and average survival rate per role for alliances of size 2.

6.2.3 Experiment 3 - 4 (`attacker`*, group), 4 (*`capitalist`*, group).* Figure 5 illustrates the results from this experiment.

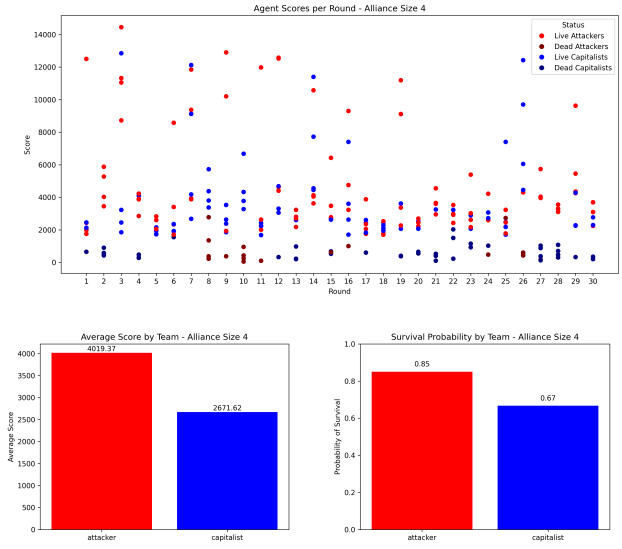


Figure 5: Individual agent scores, average score per role, and average survival rate per role for alliances of size 4.

7 DISCUSSION

7.1 BDI and MAS

As shown, the BDI paradigm could be easily used to extend the existing formalisms for multi-agent systems. In other words, with BDI the definition for a mechanism could become a lot more granular. BDI helps properly categorize the various aspects of a mechanism by grouping them into the three pillars of beliefs, desires, and intentions. Furthermore, most complex games and multi-agent environments are dynamic and partially observable. This means there is a need for a paradigm that supports constant changes. This functionality is natively supported in BDI via its belief maintenance and intention selection mechanisms. Finally, BDI is really beneficial for abstracting the agent implementation from its environment while still keeping it explainable. As illustrated, with BDI, a complex and dynamic environment could be abstracted into simple predicate logic. The user is fully in control of the extent of that abstraction and could use it for different use cases. It is worth noting that the system is also fully explainable; at any given time within the agent's execution time, the user is aware of the agent's beliefs, desires, and intentions, thus fully being able to explain its strategy and actions.

7.2 Jason Patterns

In order to implement BDI in a multi-agent context, specific patterns were utilized in the Jason implementation.

7.2.1 Message Handling. Messages are a crucial aspect of multi-agent environments. It is important to keep track of all the messages and not lose any shared beliefs since they are usually only sent once. To achieve this in Jason, various measures were taken. The plans related to message handling were not bound to an intention and were placed at the top of the AgentSpeak file. This ensures that they are handled immediately for all desires and intentions. Additionally, to ensure that the messages are properly dealt with, the handlers for them should be purely reactive and only utilize actions and beliefs. This ensures that Jason is able to run the entire block at once without potentially switching to other plans for handling the newly added desire. Here is a simple summary of this pattern:

```
+message(Type, Sender, Content): PreCondition <-
    +potential_beliefs;
    ...
    potential_actions.
...
other_plans
```

7.2.2 Synchronous vs. Asynchronous Actions. The actions for Argus can be grouped into two categories of synchronous and asynchronous, each requiring their own pattern.

- **Synchronous Actions:** these are action that can be done immediately such as `chop_wood` or `escape`. The implementation for these actions utilizes Jason's builtin `setResult` mechanism. By default Jason waits for the result of the action; thus, for these actions, the AgArch file simply talks to the Minecraft server, waits for the action to complete then set the result of the action allowing Jason to progress. It is important to ensure that the execution time for these actions is relatively quick to avoid missing reasoning cycles in Jason.

- **Asynchronous Actions:** asynchronous actions are a bit more complex since their implementation is not running on the main thread in the AgArch file. To avoid complexities, the result of such actions is bound to a belief and Jason is told to await the addition of that belief. The Java implementation however automatically sets the result of the action as true to allow Jason to proceed and account for new beliefs. A timeout is also created to avoid waiting too long on asynchronous actions. An example of such actions is `find` which is essentially a wrapper for a navigation task.

```
+!loop: searchTimeout(SEARCH_TIMEOUT) <-
    +attempted_tree_search;
    say("Looking for trees..");
    find(tree);
    .wait({+near(tree)}, SEARCH_TIMEOUT, _);
    -attempted_tree_search;
    !loop.
```

  As shown, the agent will run the `find()` action then run its internal `.wait()` action to either get a new belief of `+near(tree)` or timeout and do an action failure.

7.2.3 Action Failure Handling. As illustrated, the Jason actions can fail for a lot of reasons. An action failure results in the overall failure of its intention. Thus, it is important to properly account for action failures. The proposed pattern is adding a transient belief just before each action and removing it after the action's execution. Then simply adding a plan for the removal of the intention guarded with the transient belief to catch the action failure. For example:

```
+!loop <-
    +attempting_action;
    say("Doing something..");
    action;
    -attempting_action;
    !loop.
...
-!loop: attempting_action <-
    -attempting_action;
    say("The action failed..");
    alternate_action;
    !loop.
```

7.3 Strategy Performance

Although, the proposed strategies for Argus were quite simple, they created very interesting results. As expected, the more aggressive strategy of "attackers" was proven to be a lot more successful in collecting points and surviving the game. This is due to the fact that they were constantly going after the zombies and collecting as many points as possible; however, it is important to note that the agents with this strategy were only successful when they had allies. This is because an ally for them was one fewer enemy that could harm them. This is in contrast to the more conservative "capitalist" strategy that focused on building houses and doing self preservation. The capitalists were not really focused on attacking others and focused on survival instead. Alliances were not really beneficial for the capitalists since they are already able to survive on their own. In summary, a good strategy for this game would be the mix of both. A singular agent should have the strong survival instincts of a "capitalist" agent with the combative nature of the "attacker" agent. This type of conflicting behaviour is tricky to capture in traditional rules based agents. Perhaps, these conflicting behaviours can be modelled as the agent's "preferences" and turned into non-linear plans with the use of epistemic logic.

8 FUTURE WORKS

There are various improvements that could be applied to this work. Firstly, the game itself could be improved to support more complex functionalities. This work would be done in the form of supporting more beliefs, actions, and messages. Additionally, the data collection and experimentation could be improved to solidify the results further. Currently, each experiment was only run for 30 rounds. Furthermore, more strategies (and roles) can be explored for this game. The `attacker` and `capitalist` strategies are quite simple. More complex teams can be formed with specific roles for each member. Finally, this environment can also be used to compare the performance of Jason BDI against its the other implementations of BDI or even other paradigms. An important improvement in this area could be trying different intention selection mechanisms.

ACKNOWLEDGMENTS

This project was completed to fulfill the project requirements for COMP5900, Multi-Agent Systems. This project could not have been completed without the support of Dr. Alan Tsang and the Department of Computer Science at Carleton University.

REFERENCES

[1] Rafael H. Bordini and Jomi F. Hübner. 2005. Programming multi-agent systems in AgentSpeak using Jason. *Computational Logic in Multi-Agent Systems* 3900, 3 (2005), 143–164.
[2] Moharram Challenger, Baris Tekin Tezel, Omer Faruk Alaca, Bedir Tekinerdogan, and Geylani Kardas. 2018. Development of Semantic Web-Enabled BDI Multi-Agent Systems Using SEA_ML: An Electronic Bartering Case Study. *Applied Sciences* 8, 5 (2018). https://doi.org/10.3390/app8050688
[3] Citizens. 2025. Citizens API. https://wiki.citizensnpcs.co/API. Accessed: 2025-12-13.
[4] Mark D'Inverno and Michael Luck. 1998. Engineering AgentSpeak(L): A formal computational model. *Journal of Logic and Computation* 8, 3 (1998), 233–260.
[5] Michael Georgeff, Barney Pell, Martha Pollack, Milind Tambe, and Michael Wooldridge. 1970. The Belief-Desire-Intention Model of Agency. *Lecture Notes in Computer Science* (02 1970). https://doi.org/10.1007/3-540-49057-4_1
[6] Alejandro Guerra-Hernández, Amal Seghrouchni, and Henry Soldano. 2004. Learning in BDI Multi-agent Systems. 39–44. https://doi.org/10.1007/978-3-540-30200-1_12
[7] Jomi F. Hübner and Rafael H. Bordini. 2008. Developing a Team of Gold Miners Using Jason. In *Programming Multi-Agent Systems*, Mehdi Dastani, Amal El Fallah Seghrouchni, Alessandro Ricci, and Michael Winikoff (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 241–245.
[8] Nicholas R. Jennings, Katia Sycara, and Michael Wooldridge. 1998. A Roadmap of Agent Research and Development. *Autonomous Agents and Multi-Agent Systems* 1 (1998), 7–38.
[9] Li Liu, Shikun Chen, Huan Jin, Xiaoying Deng, Yangguang Liu, and Yang Lin. 2025. Optimizing on-demand food delivery with BDI-based multi-agent systems and Monte Carlo tree search scheduling. *Scientific Reports* 15, 1 (2025), 25083. https://doi.org/10.1038/s41598-025-10371-w
[10] Qian Long, Zhi Li, Ran Gong, Ying Nian Wu, Demetri Terzopoulos, and Xiaofeng Gao. 2024. TeamCraft: A Benchmark for Multi-Modal Multi-Agent Systems in Minecraft. arXiv:2412.05255 [cs.AI] https://arxiv.org/abs/2412.05255
[11] Minecraft. 2025. What is Minecraft? https://www.minecraft.net/en-us/about-minecraft. Accessed: 2025-12-11.
[12] PaperMC. 2025. PaperMC. https://papermc.io/. Accessed: 2025-12-13.
[13] SpigotMC. 2025. CommandHelper. https://www.spigotmc.org/resources/commandhelper.64681/. Accessed: 2025-12-13.