A Strategy Video-Game for Collaborative Agents with a Personality and Humoristic Dialogues

Jordan Mackie
Student

Alice Toniolo
Supervisor

Christopher Stone
Supervisor

***Abstract*—We built a system of collaborative agents with personality driven decisions and humoristic dialogues with the goal of providing entertainment and also providing a human friendly way of interpreting agent interactions in multiagent systems. The agents were designed to play chess and given goal-based personalities that would affect their decisions and how they would interact with other agents. We gathered feedback through allowing volunteers to play against our agents and asking questions covering usability and enjoyment. Finally, we present our results and discuss further work or areas of improvement that could be taken up.**

I. Introduction

Multiagent systems are the next step to increase the level of autonomy that can be provided by technology. Agents that are able to learn, adapt, and negotiate with other agents to achieve their goals allow for complex problems to be solved or modelled without human intervention, such as monitoring and maintaining national power grids [1].

Often, developers and users will anthropomorphise these agents when describing their behaviour. This project aims to encourage this by implementing agents with a model of personality that affects their choice of actions, by rendering the negotiations between agents in a natural language, and by using models of humour to make the interactions between agents entertaining.

Chess was chosen as our strategy game as multiagent implementations have been previously investigated, though not in a configuration where agents could have conflicting goals in the same team. However, our model for determining agent actions can be applied to any other strategy game or multiagent system.

II. Context Survey

A. Multiagent Systems in Entertainment

Many modern video games involve the user managing multiple characters to achieve some goal, such as producing in-game resources or defending against an opponent. This sort of problem lends itself easily to multiagent systems. Instead of having one artificial intelligence engine driving the actions of all the characters, developers can create agents with a limited set of actions and some concept of progress towards their goals and allow emergent behaviour to find a solution to the problem, sometimes in surprising ways.

[2] discuss how multiagent systems can be applied to create more realistic worlds in sandbox games. By implementing the environment, objects, and non-playable characters (NPCs) as agents, developers can create a world that reacts and adapts the player, but also operates in isolation from the character to provide a realistic setting. The concept of personalities is also mentioned as a way of allowing similar NPCs to exhibit different slightly different behaviours, such as having aggressive or relaxed driving styles.

Even in games with very simple rules and logic, multiagent systems can find interesting and complex solutions. OpenAI implemented hide-and-seek using agents, where hiders avoid the line-of-sight of the seekers [3]. The game was played in a world with randomly generated walls and objects such as ramps (for climbing over walls) and blocks (for forming barricades). What made this fascinating is that the agents were not incentivised to use these objects, but after repeatedly playing and learning, both the hiders and seekers created strategies such as blocking each other in a safe area, and even removing the objects from the other team before hiding.

Knowing that multi-agent autocurricula can realise strategies not considered by humans, [4] discuss how they can be applied to strategy board games such as Diplomacy and Risk. They describe a generic framework for supporting agent-based competitive bots for board games and then implement bots for the aforementioned games. Their research suggests that for games with a large number of units and a large action space, a multi-agent approach can identify effective strategies quicker than the exhaustive methods used for chess engines.

B. Multiagent System Frameworks

Most multiagent systems have similar requirements, and so several frameworks have been developed to bootstrap their development. [5] provide a very thorough survey of the frameworks currently in use today and highlight various features and drawbacks in their implementation.

The most popular is the Java Agent Development Framework (JADE) [6]. It conforms to the FIPA standard, which is a protocol for agent communication that involves defining performatives (e.g. request, inform), language name, and other message meta-data during communication. The benefit of systems that abide this standard is that they are able to interact regardless of the technology used to implement them. JADE provides other important features such as base classes for agent functionality called behaviours, a directory facilitator (DF) agent implementation to allow agents to find each other, and an Agent Management Service that manages and tracks

the lifecycle of agents and allows them to move between containers (which can exist on multiple hosts). To aid during development, JADE also incldues a GUI for debugging and manually interacting with agents.

Other frameworks provide APIs that encourage certain design patterns. For example, Jason was built to support the belief-desire-intention (BDI) design model which attempts to seperate the processing of choosing a plan from the execution of the chosen plan [7]. Jason provides many of the same functionalities as JADE, but the latter was chosen due to being slightly more flexible.

C. Models of Personality and Emotion

Creating a truly immersive video game requires characters that the player can empathise with. Robots have been shown to be able to influence human behaviour as an authority figure [8] and when begging not to be turned off [9] by expressing emotions. [10] created and demonstrated a model of personality and emotion that would allow agents to react differently to the same stimulation. For example, when a agent with an 'introverted' personality is offered help, they are less likely to accept it due to the prolonged interaction it would entail.

[11] also created a model that would use personality, emotion, and social relationships to determine the behaviour of NPCs in a video game. The frequency and tone of interactions between NPCs as well as the NPC and the player were accounted for when choosing facial expressions and tone of voice during conversations. Test subjects described feeling especially attached to the NPCs that utilised this model.

A useful aspect of multiagent systems is that agents can be developed in isolation but still interact (e.g. an agent that searches for cheap transport options and an agent responsible for auctioning train tickets could be developed separately with no knowledge of the logic being used by the other). [12] discuss how developing heterogeneous agent systems using personalities and social structures could help when dealing with third-party agents that have been constructed to lie and exhibit selfish or uncooperative behaviour.

D. Collaborative Argumentation

Knowledge is distributed in a multiagent system therefore specialist agents need to be able to alter the beliefs of others by appealing to their individual goals. [13] implemented a framework that achieves this based on a scheme given by [14]:

> In the current circumstance R, we should perform action A, which will result in new circumstances S, which will realise goal G, which will promote some value V.

Specifically, they were able to create a framework for multiple parties to discuss and collaborate which could greatly affect the design of multiagent systems that utilise it. By producing an ontology which any agents involved in the discussion understand, goals and circumstances can be conveyed through the use of predicates and concepts. Conveniently, JADE already provides an API for creating ontologies. However, the manual work required to construct an ontology that captures the components and logic of a game of chess was found to be far too cumbersome. While this would be the most interoperable solution, serializable Java objects were used instead for quicker development.

The overhead of argumentation required in multiagent systems can be a quick filter to determine which problems it is a suitable solution for. For example, [15] investigated how quickly agents that used collaborative argumentation to achieve global consistency in a time-constrained task (i.e. escaping a burning building) performed. Their results suggest that optimisations or different approaches to the protocol design would likely be necessary for a real-time strategy game, but it could depend on many factors such as the number of agents, the distribution of knowledge, and more.

Previous work has shown that multiagent chess systems do not generally perform well strategically compared to normal tree based models for chess if the agents are given any limitations that restricts their knowledge. [16] limited pieces such that they could only evaluate their immediate surroundings, and all pieces would reach a consensus based which focused on maximising captures and minimising losses. Our solution will not limit the agents knowledge of the board, but pieces will still have to reach some consensus on which move to take based on their own individual values.

E. Natural Language Generation

In order to make the discussions between agents more entertaining and user-friendly to observe, our project involves using natural language generation (NLG) methods to translate the messages or agent intent to plain English. [17] built an NLG system for a particular ontology language (W3C Ontology Language a.k.a. OWL), which is able to construct texts corresponding to objects and their properties in the ontology in English and Greek. [18] achieved similar results for one of the other Semantic Web Formalisms (specifically RDF) and were able to produce instructions for cooking recipes from a corpus of data in a far less human-friendly format. They were even able to adjust the level of technical jargon in the resulting text to accommodate for the readers familiarity with cooking. The ontology allowed for optimised searching of the parse trees but they also required a domain corpus (i.e. existing textual recipes) in order to properly train their natural language generator. Their solutions are specific to the Semantic Web project, and would not produce "conversations" like we are planning to do, but provide a good foundation for implementing this functionality.

Generic frameworks have also been created that are not based strictly on ontologies which were also considered. For example, SimpleNLG [19] is an NLG engine that allows for valid English phrases to be constructed through a very simple Java API. However, after some time experimenting with this library, it was realised that it was very difficult to reliably produce a variety of phrases, though it was useful for controlling features such as grammatical person and tense.

[20] created a robot capable of providing dialogue while playing against a human at tic-tac-toe, but used a slot-filling technique based on templates that suffered from little flexibility and left no room for personality in dialogue creation. Contrarily, systems such as Siri and Alexa are examples of successful dialogue systems that have been able to provide assistance to humans in the form of natural language conversations by avoiding the template based approaches.

These systems instead take advantage of machine learning techniques such as deep recurrent neural networks (DRNN). [21] surveyed projects that applied these modern techniques for NLG, and concluded that in order to account for aspects such as current state of the program during the execution of conversation as our project requires, a hybrid of the machine learning and rule based approaches may be necessary.

The dynamic finite-state machine implementation of a chatbot called Tartan [22] was also especially useful for the project, as it focused on natural conversation flows rather than the extraction of information from a human user like most other chatbot research. Their conversation state machine design was combined with the RiTa library's RiGrammar (a probabilistic context free grammar) in order to produce a variety of phrases and conversation flows.

F. Computational Humour

Simply reading the discussion between agents would certainly not make for an entertaining game, and so adjusting the dialogue or agent behaviour is required. However, artificial humour also has benefits for everyday computing: a smartphone that is able to sympathise with the user when it is unable to connect to a weak WiFi network or cheer them up with a joke when they miss their bus is one that the user is more likely to enjoy using.

Unfortunately, given humour is an incredibly contextual and culture driven concept, it is no surprise that computational humour is well researched yet still not 'solved'. Jason Rutter describes why using artificial intelligence for humour is particularly challenging:

> "Humour is a very interesting way to look at artificial intelligence because at some point something has to have two meanings, which is not easy to do with a computer." [23]

There are three main theories of humour that are used for computational humour: superiority theory (laugh about the misfortune of others), relief theory (using taboo subjects to release tension), and incongruity theory (using lexical and structural ambiguity). The last theory is currently the most popular, and was used during the development of JAPE [24]. JAPE identifies features such as homophones to produce jokes like:

> "What is the difference between leaves and a car? One you brush and rake, the other you rush and brake."

However, JAPE does not allow for much user interaction, and most textual humour generation takes the form of simple puns and riddles. Our agents could instead generate situational humour by surprising the user by diverting suddenly from expected behaviour. The Suslov Model of humour accounts for the fact that we subconsciously predict expected conclusions to situations and phrases, and that contradictions in what is the most likely direction for a conversation or situation to what was previously predicted can create a humour response [25].

These probabilistic models currently appear more hopeful than the rule based models used by older implementations such as JAPE. Instead, [26] used an existing corpus of jokes and a recurrent neural network to produce a system capable of generating jokes and even anti-jokes (jokes without an actual punchline).

III. System Design

An existing multiagent development framework was chosen instead of rolling our own in order to abstract away matters such as message delivery and agent life-cycle management. As mentioned earlier, JADE was chosen specifically due to its flexibility and its extensive coverage of the FIPA standard that could allow our agents to interact with other implementations with ease.

With the intention of possibly serving the game over the internet once complete, the system had to be designed to allow for communication with agents from outside of the agent platform. The 'JadeGateway' API was used with a Spring Boot server and React UI to to achieve this. Figure 1 provides a high level overview of the technologies at each layer.

Fig. 1. Technologies used at each layer.

A. Front-End

React [27] was chosen due to it being the most familiar web framework for the developer of this project, and because a component already existed for rendering chessboards [28]. React uses a virtual DOM in order to efficiently re-render components based on changes in state. For example, when a new message arrives, the state of the chat transcript component is updated to contain the new message which triggers an update to only the chat transcript and its children that are also affected.

As the project grew, Redux [29] was introduced in order to help with state management of components. Redux provides a single global state which components 'connect' to in order to receive updates. Updates are 'dispatched' as actions, which are then passed through a pipeline of reducers which update parts of the global state depending on the action and its payload. Middleware can also be introduced to allow certain actions to create chains of updates (e.g. making an asynchronously API call, then dispatching another action when the response is received), but the reducers themselves are always pure

functions. This functional approach to managing components made the project easier to test and errors easier to trace and correct. Figure 2 shows the parts of the state that are managed by each reducer.

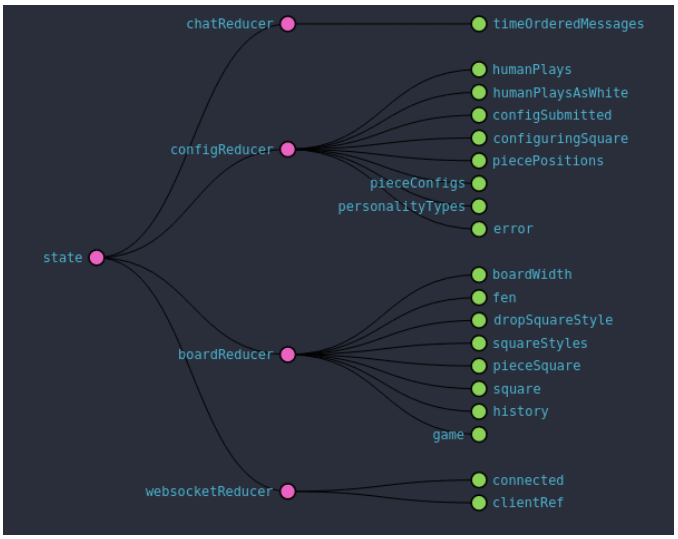

Fig. 2. Redux reducer tree showing which parts of the state are managed by which reducers.

One of the challenges faced were that the abstraction of the chessboard component provided by a third-party library did not allow for atypical visual elements such as adding name tags to each piece or dialogue boxes. This then required manually implementing an overlay that would draw another chessboard grid, track positions of pieces, and draw the name tags and dialogue boxes in the correct locations.

The UI is composed of two main views: configuration and gameplay. The configuration view (Figure 3) allows the user to choose if they want to play against the agents or watch two sets of agents play against each other. By clicking on the pieces they will not be controlling, the user can also name the pieces and define their personality type.

On submit, any unconfigured pieces are filled out at random from the set of available personalities and a set of names. The game configuration is then sent as a POST request to the gateway server, which verifies and creates the resources necessary for the game and returns its ID to allow the client to publish and subscribe to game events at the correct web socket endpoint.

Figure 4 shows the sequence diagram for how resources are created once the user submits the game configuration

The game view (Figure 5) also shows the game board with name tags over the corresponding pieces and any dialogue next to the speaking piece. The chat and move history is also rendered in a scrolling transcript at the bottom of the page in alternating colours for clarity. When hovering over pieces the user is able to see all currently available moves to them. Moves are verified client side by the chess.js [30] engine before either being rejected or sent to the server. Figure 6 shows how human moves are broadcast to all subscribers.

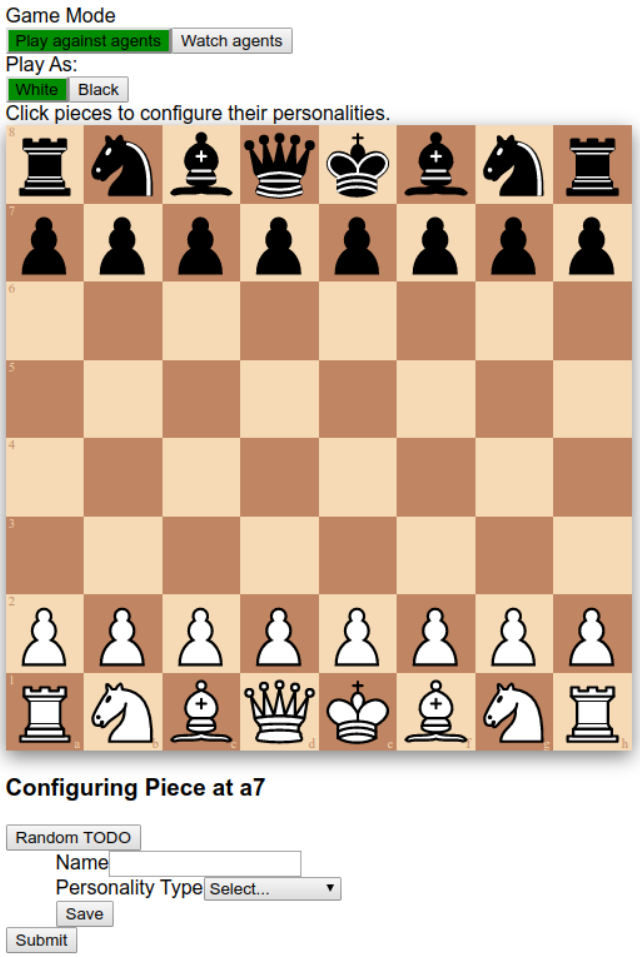

Fig. 3. Configuration view.

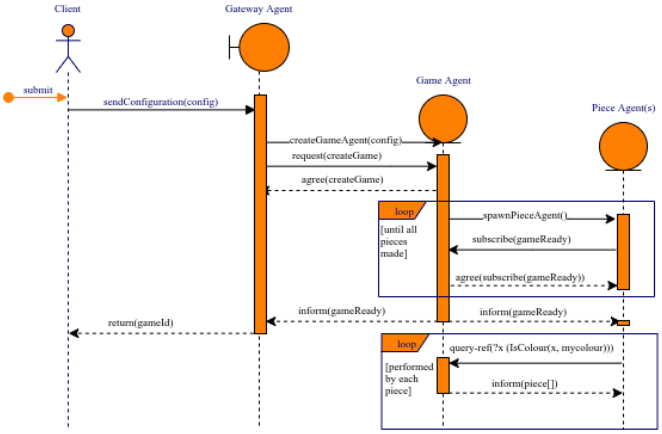

Fig. 4. Game creation sequence diagram.

B. Agent Gateway Server

The gateway server has several purposes:

- Serve the web app
- Handle game resources
- Route messages between users and relevant agents

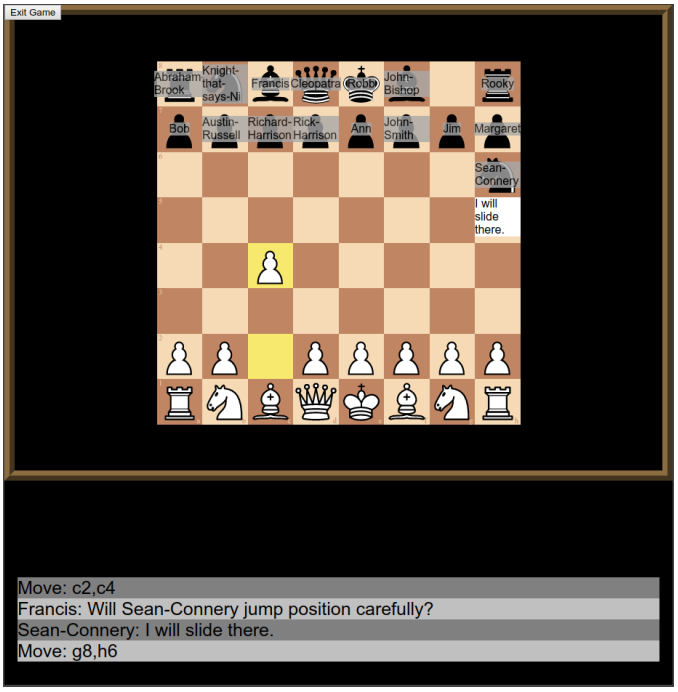

Fig. 5. Game view.

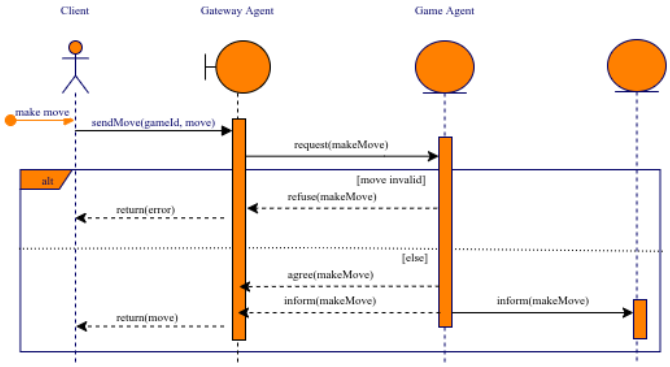

Fig. 6. Human move sequence diagram.

JADE does have support for agents serving web resources directly, but the JadeGateway API was used instead as it was simpler to set up. The JadeGateway API effectively creates an Agent for the server, which can then send units of functionality called 'Behaviours' to be executed by the agent (e.g. sending messages to the agents within the platform). This could potentially be a bottleneck if the number of users of the server was to grow large enough due to agents being single-threaded by default, so an implementation at scale would likely need to change this interface. For the purposes of this project however, it was more than sufficient.

On start-up, the server uses the details in a configuration file to connect to the JADE platform. It also exposes a limited set of HTTP endpoints to allow the client to create games and to query available personality types. An in-memory message broker is used for handling websocket sessions once a game is set up so that clients can subscribe to chat and move messages, and send their own moves.

The JadeGateway polls for incoming messages using the ListenForGameAgentMessages behaviour, which is provided a message translator and a message handler for translating ACLMessages from agents to a data model the server can work with and passing the message to the correct handler. This behaviour allowed the server to utilise JADEs scheduling system which would only run the behaviour if a new message was received, instead of busy waiting for incoming messages.

C. Jade Agent Platform

Finally, the agent platform is where all of the agents are hosted. Here we will give a rough overview of how JADE agents schedule their functions, while in section IV we discuss the actual logic that drives our agents.

JADE agents are single-threaded by default, and execute behaviours according to the flow diagram shown in Figure 7. The basic JADE behaviour implements the methods shown in figure 7, but the framework also provides some helper implementations such as the FSABehaviour which was used heavily during this project.

Behaviours are scheduled such that they are executed repeatedly round-robin until either they are complete (done() returns true) or block() is called during the action() method. The latter puts the behaviour into a waiting queue so that it is only triggered again once the agent receives a message.

Behaviours can filter the messages that are received using the MessageTemplate class, which uses pattern matching to only allow for messages containing features such as a specific performative (e.g. REQUEST, PROPOSE) or a specific conversation ID. This ensures that behaviours do not consume messages that were not intended be consumed by them.

IV. Agent Implementation

Our implementation involves two different types of agent: game agents and piece agents. Game agents are responsible for maintaining the state of the game and initialising the piece agents, and piece agents represent pieces on the board, each trying to achieve their own individual goals.

A. Game Agent

The game agent upon creation waits for a request to begin a chess game. It also answers any queries regarding the current status of the game and information about the pieces being represented by agents. The FIPA standard protocols for agent interaction [31] were followed so that the game agent could provide a consistent API for any implementation of piece agents. For example, the request interaction is used for triggering moves as shown in figure 6.

Requests for information are semantic logic expressions with variables which the game agent then populates in its reply. For example, agents wishing to receive updates whenever a new move is made will send a message with a SUBSCRIBE performative and the expression:

$$(\ iota\ ?Move\ (\text{"}MoveMade\text{"}\ ?Move)) \tag{1}$$

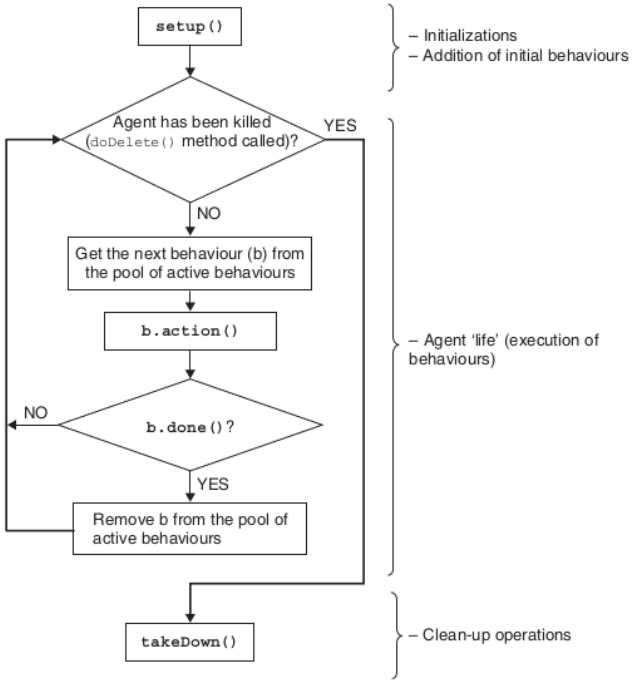

Fig. 7. JADE Agent behaviour scheduling and life-cycle [6].

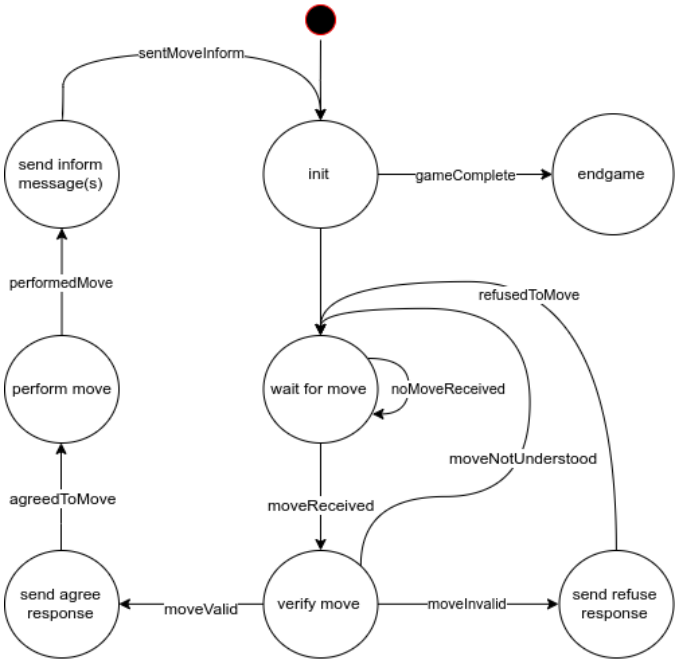

Fig. 8. Game Agent game handling FSA.

This will then be replied to every time a new move occurs with the variables given values, for example when a piece moves from B8 to C6:

$$((= \ (iota \ ?Move \ (\text{"}MoveMade\text{"} \ ?Move)) \\ (Move \ : Source \ (Position \ : Coordinates \ B8) \\ : Target \ (Position \ : Coordinates \ C6)))) \quad (2)$$

Once the game has started, the game agent adheres to the FSA shown in figure 8.

The actual chess logic is managed by a third party library [32]. While this saved a lot of time by providing a way of quickly validating moves, generating valid moves, and tracking the board state, it suffered from the same problem as the chessboard component used by the front-end: it did not provide a way to differentiate between each piece of the same type. This resulted in a wrapper class being implemented which would maintain the mapping between the set of our custom piece representation and the third party board pieces. In hindsight it was clear that it would have been simpler to have implemented the chess logic ourselves, but the wrapper solution was sufficient.

B. Piece Agent

The piece agent implementation is the focus of this project. They are initialised with their piece type, colour, personality type, and starting position on the board. They are notified by the game agent once the game is ready and whenever a move is made successfully either by the human or another agent.

1) Game State: The GameState class wraps our chessboard wrapper to provide another layer of abstraction in order for the state of the board to be an immutable object. This was done to avoid complicated undo logic when agents wanted to 'test' moves and evaluate the board after a move is made. Instead, when the applyMove(PieceMove move) function is called, a copy of the board is made. This involved copying the set of alive and captured pieces, as well as updating the piece objects themselves if any changed position.

In order to reduce the amount of objects being created and destroyed by this process, piece instances would only be copied if they had changed state in some way (captured/changed position).

2) Personalities and Reasoning: Agent personalities were designed based on the goal-based trait/value model defined by [12]. Figure 9 shows the UML class diagram for our implementation. The personality is initialised with a set of traits (e.g. shy, aggressive) which each have a set of values. Values define what the agent will interpret as good and bad moves based on some criteria (avoid other pieces, maxmise captures). The value class is abstract and is extended by implementations that when given a witnessing piece, the current game state, and a move, are able to produce a response to that move.

This implementation allows different traits to share goals. The move most preferred by an agent is the one that appeals to the most of it values, and so consensus can be reached by finding the move that appeals to the most pieces, even if they have different values.

3) Single Thread of Conversation Protocol: In order to make the conversation between agents seem natural and remain easy to follow, a protocol had to be established to ensure that one agent would speak at a time. The project was initially

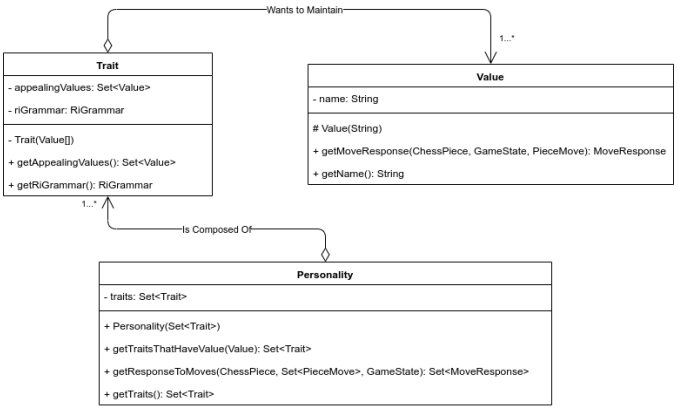

Fig. 9. UML Diagram of Agent Personalities.

constructed with conversation flow dictated strictly by a finite state machine. This solution was incredibly restrictive as it required providing a transition path for any conversation path that we wanted to support. After reviewing the work by [22] and finding that this strict FSA approach was inherently detrimental to producing natural conversation flows, the original design was thrown out and replaced with the simpler and far more flexible model shown in Figure 10.

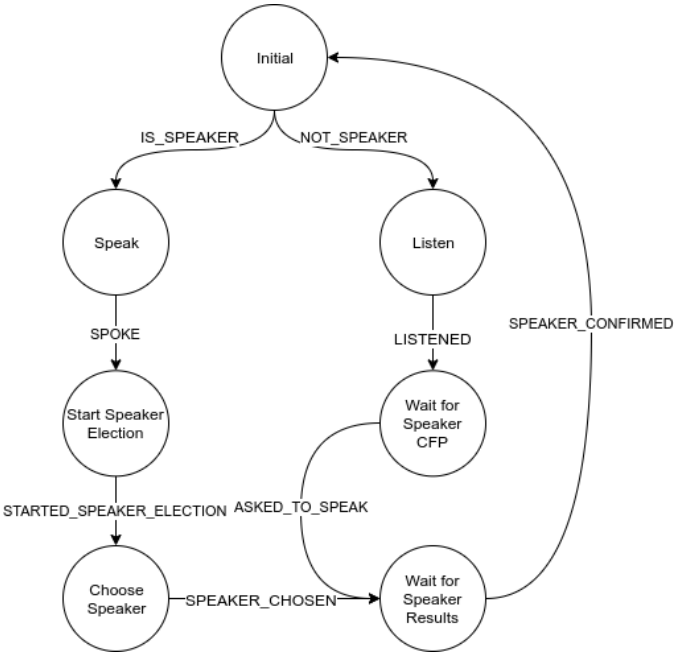

Fig. 10. Conversation protocol FSA

The conversation FSA designed by [22] was for a human-agent chatbot, and so was adapted to include a token-passing protocol to determine which agent would speak next. One of the benefits of this design is that it allowed for the agents to chat throughout the game, not just during their turn. This meant that the agents could provide ambient conversations and respond to events like the previously made move while the human player chose their next move.

At the start of each turn (i.e. when a new move is received or the very first turn of the game), each agent returns to the initial state to ensure that they are all synchronised. The king is assumed first to speak. The other pieces listen for any messages from the speaker and the speaker chooses what to say. Once sent, the speaker begins the new speaker election by sending a call for proposal (CFP) message to all alive pieces (and themselves), and waits until every pieces has submitted a proposal to be the next speaker. Finally the speaker chooses from the set of proposals, sends a rejection to all but the chosen speaker, and then the cycle begins anew.

The requirement that all agents reset to the initial state at the start of each turn was necessary to avoid race conditions during the speaker election. For example, if a move was received that resulted in one of the agents being captured, that agent would immediately remove themselves from the game. However, if that agent was the speaker during an election, then all other agents would be stuck waiting for the next speaker to be chosen. Whilst other solutions would allow for the conversation flow to continue, it was deemed much simpler to reset the conversation FSA during each turn.

As a side effect of this decision, messages such as the speaker election CFP could be waiting in an agents message queue if they reset before receiving it. To avoid this resulting in a loss of synchronisation between agents, a new conversation ID is assigned during each cycle of the FSM which is composed of the turn ID (i.e. how many moves have been made so far) and the round ID (how many cycles of the conversation FSA have occurred this turn). The conversation ID is used to filter incoming messages to ensure that irrelevant messages are ignored. Unfortunately this could result in a build-up of expired messages over time, but given most chess games were around 80 moves [33] and roughly 5 messages per move would be lost in the queue, the leak would not be substantial. Especially when we considered that the messages would be garbage collected once the agent representing the piece was removed from play.

The turn and speaker rotation counters is provided by the ConversationContext class, along with the current speaker agents AID (Agent Identifier) and the conversation planner.

4) Conversation Planner and Argumentation: Each agent maintains a conversation planner which tracks what moves have been discussed so far, and produces a response for this agent once it has its turn to speak. Figure 11 shows the classes used to maintain this information.

Whenever an agent enters the speaker state shown in figure 10, they call the produceMessage() function to choose their next action. If this action involves making a move, they'll queue a behaviour to send the move to the game agent. Regardless, they send their chosen conversation message to all the other agents (including themselves).

Listener agents receive the conversation message and record it by calling the handleConversationMessage(ConversationMessage) method. The conversation planner delegates storing the message to the current turn discussion (the head in the list of discussions).

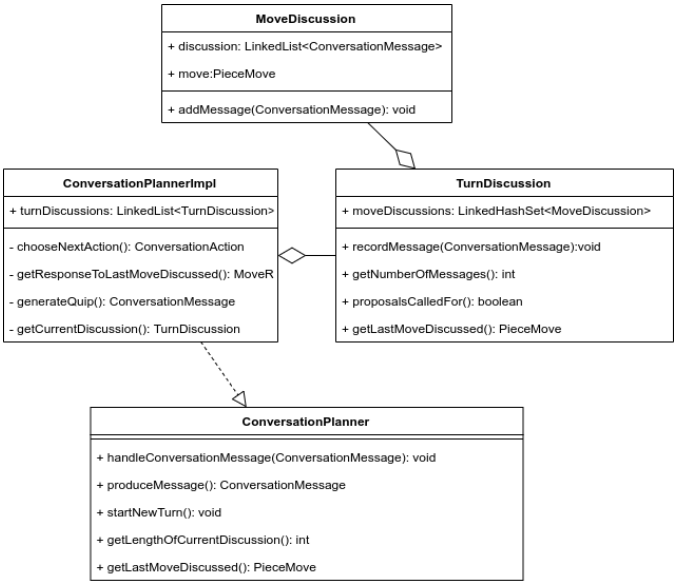

Fig. 11. Move Discussion Class Diagram

The turn discussion will fetch the move discussion for the move in the given conversation message or create a new one if one doesn't already exist and add the message to the relevant move discussion. Once the message is added to the move discussion, the discussion is removed and re-added to the set of move discussions in order to move it back to the top of the converstation 'stack' (since insertion order is maintained by Java's LinkedHashSet implementation).

By maintaining the order that moves have been discussed, agents are able to create arguments that better resemble human conversations by referring to previous discussions. [22] uses a similiar technique to allow their chatbot to regress to previous topics of discussion once the user gives some kind of terminating statement to the current topic conversation.

A move discussion with a null move is used to represent a request for move proposals, which is sent either to start up the argumentation process or when an agent is not satisfied with the move currently being discussed but is does not feel strongly about any moves themselves.

The output of chooseNextAction() in the ConversationPlannerImpl is the next conversation action to be taken. These actions are essentially transitions which control the flow of the conversation and are based off the interruption FSMs in Tartan [22]. Table I lists examples of the conversation actions possible. The actions prefixed with wildcards can be used in conjuction with other actions. For example, an agent can voice opinion with justification, or voice their dislike and propose an alternative.

Each of these actions are implemented as classes that extend the ConversationAction class shown in Figure 12, and once performed they return a ConversationMessage instance with all the information needed for other agents to update their own understanding of the conversation context. The actions can also extend each other if their affect on the conversation

TABLE I
CONVERSATION ACTIONS WITH VERBALISED EXAMPLES.

Conversation Action	Example
Acknowledge	"Ok."
Ask for Proposals	"What should we do next?"
Perform Move	"Going to move to A5."
Propose Move	"What about B4 to B3?"
Revisit Move	"Let's go back to talking about the move to G7."
Voice Opinion	"I like that idea!"
* Propose Alternative	"but moving to G3 would do that too."
* with Justification	"because it would capture an enemy."
Quip	"I'm a bit scared."

TABLE II
CONVERSATION MESSAGE FIELDS AND THE AGENT INTENTION.

Field	Predicate	Agent Intention
response	== null	Requested proposals
response.opinion	!= null	Voiced opinion
response.reasoning	!= null	Gave justification
response.alternativeProposition	!= null	Proposed alternative
response.performed	== true	Performed move

flow is the same but the natural language representation needs to differ (for example, revisiting a previously discussed move and proposing a new move both begin a new move discussion but would be expressed differently in english).

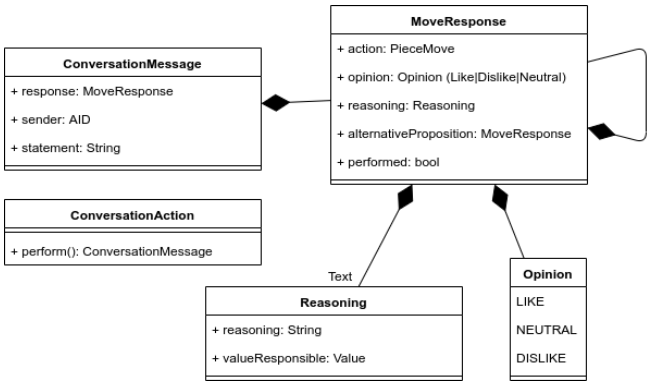

Fig. 12. Conversation Message Class Diagram

Conversation messages contain the name of the piece that sent it, the natural language representation of what the agent wanted to communicate, and a move response. The mapping between each field and agent intentions are described in table II. The alternative proposition field starts a new move discussion if present and the same logic is applied recursively as though it was the the move response in the original conversation message.

This information is enough to allow agents to determine how other agents feel about the move being discussed and make decisions about what conversation actions would logically follow, but the contents of the statement in the Conversation-Message are completely flexible.

The next conversation action is chosen by building up the set of responses that would make sense given the discussion so far and then selecting one from the set randomly. Most responses are equally likely, but some are given lower likelihoods of

occuring (such as performing a move) in order to extend conversation and encourage discussion rather than agents just repeatedly proposing their own preferred moves.

Choosing responses that 'made sense' involved checking the current discussion context and determining the following:

- Is this the first message this turn?
- Did the previous message ask for proposals?
- Have moves been discussed already?
- What is this agents opinion of the move currently being discussed?
- Does a move exist that I can perform?

In order to create the illusion that pieces are moving themselves, only the speaker can peform a move and that move has to involve the same piece. This choice will likely result in less strategic moves being made (for example, a piece will not want to put itself at risk, even if it means saving a more important piece), but makes for a more entertaining narrative.

The reasoning field in move responses are populated by the Value of an agent assessing a given move. This gives a natural language explanation for the value of the opinion field in the move response, and the value that was used to compute it. This allows other agents to apply the same value to all other moves (even if they do not have that value themselves) and provide a compromise that also appeals to their own values.

This argumentation feature could be improved by expressing the value as a predicate that other agents could unpack instead of being tied to the Value classes in our chess agent library.

5) Natural Language Generation:

V. Evaluation

VI. Future Work

VII. Conclusion

References

[1] Thies Wittig, editor. *ARCHON: An Architecture for Multi-agent Systems*. Ellis Horwood, Upper Saddle River, NJ, USA, 1992.

[2] Sergio Ocio and José Antonio López Brugos. Multi-agent systems and sandbox games. 2009.

[3] Anonymous. Emergent tool use from multi-agent autocurricula. In *Submitted to International Conference on Learning Representations*, 2020. under review.

[4] Stefan J. Johansson. On using multi-agent systems in playing board games. In *Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems*, AAMAS '06, pages 569–576, New York, NY, USA, 2006. ACM.

[5] Kalliopi Kravari and Nick Bassiliades. A survey of agent platforms. *Journal of Artificial Societies and Social Simulation*, 18(1):11, 2015.

[6] Fabio Bellifemine, Federico Bergenti, Giovanni Caire, and Agostino Poggi. *Jade A Java Agent Development Framework*, volume 15, pages 125–147. 01 2005.

[7] Rafael Bordini, Jomi Hbner, and Michael Wooldridge. *Programming Multi-Agent Systems in AgentSpeak Using Jason*, volume 8. 10 2007.

[8] Derek Cormier, Gem Newman, Masayuki Nakane, James Everett Young, and Stephane Durocher. Would you do as a robot commands ? an obedience study for human-robot interaction. 2013.

[9] Aike C Horstmann, Nikolai Bock, Eva Linhuber, Jessica M. Szczuka, Carolin Straßmann, and Nicole C. Krämer. Do a robots social skills and its objection discourage interactants from switching the robot off? In *PloS one*, 2018.

[10] Arjan Egges, S Kshirsagar, and Nadia Thalmann. A model for personality and emotion simulation. pages 453–461, 01 2003.

[11] Andry Chowanda, Peter Blanchfield, Martin Flintham, and Michel Valstar. Computational models of emotion, personality, and social relationships for interactions in games. 05 2016.

[12] Cristiano Castelfranchi, Fiorella de Rosis, Rino Falcone, and Sebastiano Pizzutilo. Personality traits and social attitudes in multiagent cooperation. *Applied Artificial Intelligence*, 12:649–675, 1998.

[13] Elizabeth Black and Katie Atkinson. Dialogues that account for different perspectives in collaborative argumentation. pages 867–874, 01 2009.

[14] Katie Atkinson and Trevor Bench-Capon. Practical reasoning as presumptive argumentation using action based alternating transition systems. *Artificial Intelligence*, 171(10):855 – 874, 2007. Argumentation in Artificial Intelligence.

[15] Gael Hette, Gauvain Bourgne, Nicolas Maudet, and Suzanne Pinson. Hypotheses refinement under topological communication constraints. 01 2007.

[16] Henric Fransson. Agentchess an agent chess approach-can agents play chess ? 2003.

[17] Ion Androutsopoulos, Gerasimos Lampouras, and Dimitrios Galanis. Generating natural language descriptions from owl ontologies: The natural owl system. *J. Artif. Int. Res.*, 48(1):671–715, October 2013.

[18] Philipp Cimiano, Janna Lüker, David Nagel, and Christina Unger. Exploiting ontology lexica for generating natural language texts from RDF data. In *Proceedings of the 14th European Workshop on Natural Language Generation*, pages 10–19, Sofia, Bulgaria, August 2013. Association for Computational Linguistics.

[19] Albert Gatt and Ehud Reiter. SimpleNLG: A realisation engine for practical applications. In *Proceedings of the 12th European Workshop on Natural Language Generation (ENLG 2009)*, pages 90–93, Athens, Greece, March 2009. Association for Computational Linguistics.

[20] H. Cuayhuitl. Deep reinforcement learning for conversational robots playing games. In *2017 IEEE-RAS 17th International Conference on Humanoid Robotics (Humanoids)*, pages 771–776, Nov 2017.

[21] Leire Ozaeta and Manuel Graa. *A View of the State of the Art of Dialogue Systems*, pages 706–715. 06 2018.

[22] George Larionov, Zachary Kaden, Hima Dureddy, Gabriel Kalejaiye, Mihir Kale, Srividya Potharaju, Ankit Shah, and Alexander Rudnicky. Tartan: A retrieval-based socialbot powered by a dynamic finite-state machine architecture. 12 2018.

[23] Will Knight. Computer crack funnier than many human jokes. https://www.newscientist.com/article/dn1719-computer-crack-funnier-than-many-human-joke/, 2001.

[24] Graeme Ritchie, Ruli Manurung, Helen Pain, Annalu Waller, Rolf Black, and Dave Mara. A practical application of computational humour. *Proceedings of the 4th International Joint Workshop on Computational Creativity*, 01 2007.

[25] I. M. Suslov. Computer model of a "sense of humour". i. general algorithm, 2007.

[26] Abhinav Moudgil. Humor generation with recurrent neural networks. https://amoudgl.github.io/blog/funnybot/, 2017.

[27] React a javascript library for building user interfaces.

[28] Will B. chessboard.jsx. https://github.com/willb335/chessboardjsx, 2019.

[29] Redux a predictable state container for js apps.

[30] Jeff Hlywa. chess.js. https://github.com/jhlywa/chess.js, 2019.

[31] FIPA. Fipa interaction protocol specifications. http://www.fipa.org/repository/ips.php3 , 2002.

[32] Ben-Hur Carlos Viera Langoni Junior. chesslib. https://github.com/bhlangonijr/chesslib, 2019.

[33] @ebemunk. A visual look at 2 million chess games. https://blog.ebemunk.com/a-visual-look-at-2-million-chess-games/, 2016.