Implementation of a Dialogue Game for Persuasion Over Action

Katie Atkinson, Trevor Bench-Capon and Peter McBurney
Department of Computer Science,
University of Liverpool,
Liverpool L69 3BX, UK
{k.m.atkisnon, tbc, p.j.mcburney}@csc.liv.ac.uk

Abstract

This report contains details of the implementation of a dialogue game protocol, named the PARMA (Persuasive ARgument for Multiple Agents) protocol, realised in the Java programming language. The protocol embodies an earlier theory by the authors of persuasion over action. This theory enables participants to propose, defend, and attack an action, or course of actions and the implementation allows two human participants to engage in a computer mediated dialogue, in accordance with the theory.

1. Introduction

In [5] we have previously presented a theory of rational persuasion over action where the proponent of an action attempts to persuade another party to endorse this particular action (or course of actions). We have gone on to implement this theory in the form of a dialogue game written in the Java programming language, which we will describe in detail in this report. In [1] we have previously specified the precise locutions for stating, attacking and asking about a player's position, according to our protocol. This specification also gives details of the exact pre and post-conditions associated with the players' commitment stores, for each of the locutions set out in this specification, the details of which can also be found in [1].

This report is structured as follows: Section 2 contains the analysis of the problem in the form of a primitive class diagram. Section 3 gives details of the design for the dialogue game, presented in the form of simplified UML style diagrams and tables presenting summaries of fields, methods and constructors used in all classes. Section 4 presents a simple state transition diagram showing all the possible types of moves that can be made at any given stage and this in turn leads to changes in the roles of the participants, with respect to which player is assigned to the role of speaker and which player is assigned to the role of hearer. Section 5 provides an evaluation of the implemented game and a discussion of issues that have come to light through implementing the game. Finally, Section 6 presents the conclusions and possibilities for future work.

2. Analysis

Figure 1 below shows a primitive class diagram showing all the classes that are used to implement the dialogue game, which embodies the PARMA protocol. The classes are all represented in the form of simplified UML style diagrams.

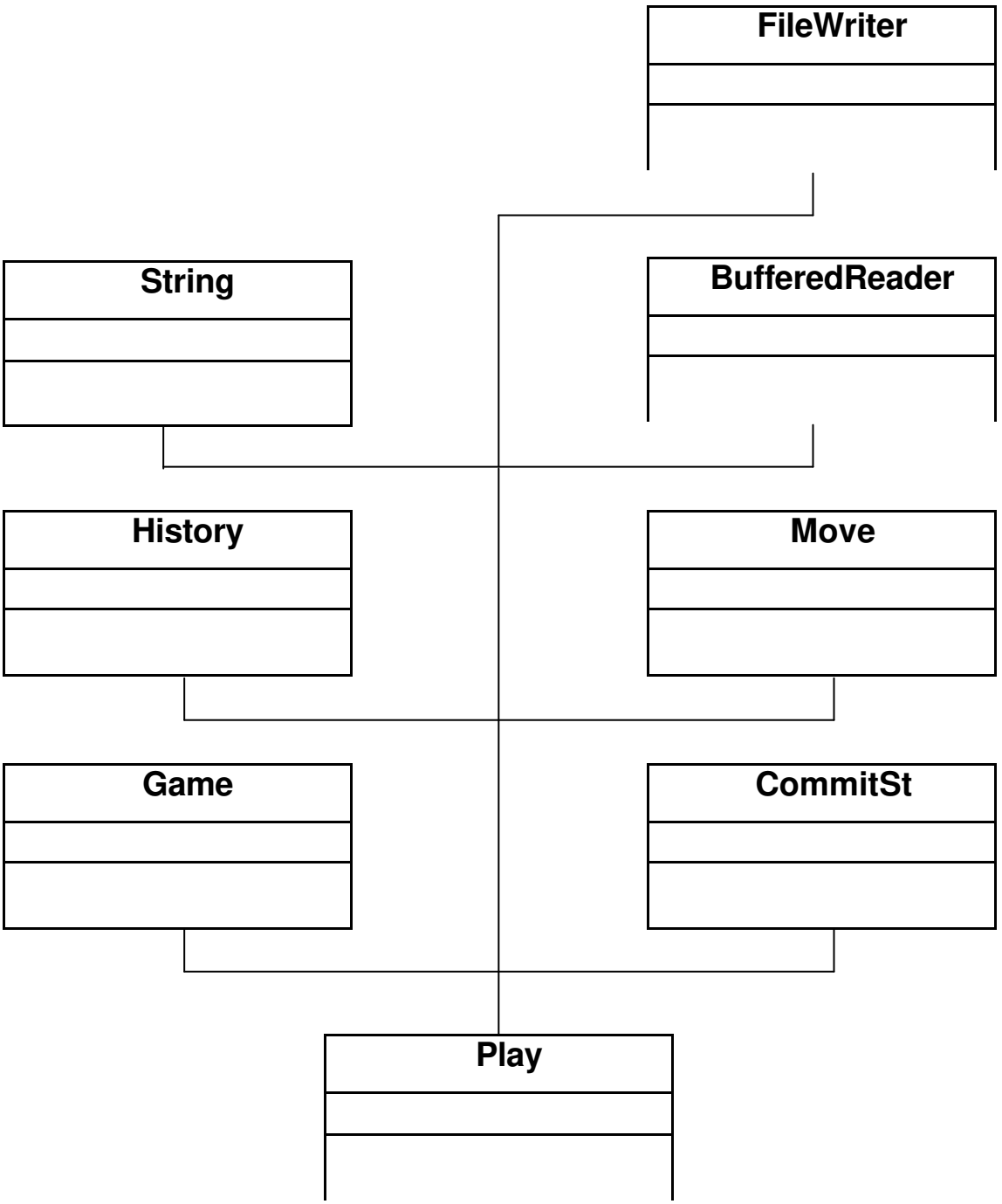


Figure 1 - Primitive Class Diagram.

Given below are details of each of the individual classes shown in Figure 1. Each of the UML style diagrams representing a class shows the fields, constructors and methods that are used in each individual class.

2.1 History Class

History
private String[][] history
public History() public void updateHistory(String, String) public void legalUpdate(String) public void illegalUpdate(String) public void printHistory()

2.2 Move Class

Move
private boolean movePossible private String[] possibleMoves private static final int LOWER_BOUND private static final int ARRAY_LENGTH
public Move() public void checkPossible(String, String, String, String, String, CommitSt, Game, History, Move) public void successful(String, String, String, String, String, CommitSt, Game, History, Move)

2.3 Game Class

Game
private String speaker private String hearer private String[] possibleMoves public static BufferedReader keyboardInput
public Game() public void firstMove(String, String, CommitSt, Game, History, Move) public void turnFinished(String, String, String, String, String, CommitSt, Game, History, Move) public void makeMove(String, String, String, String, String, CommitSt, Game, History, Move)

2.4 CommitSt Class

CommitSt
private String[][] play1ComSt private String[][] play2ComSt private String content public static BufferedReader keyboardInput
public CommitSt() public void answerAsk(String, String, String, String, String, CommitSt, Game, History, String, String, Move) public void askAccept(String, String, String, String, String, CommitSt, Game, History, String, Move) public void illegalMove(String, String, String, String, CommitSt, Game, History, String, History, String, Move) public void legalAccept(String, String, String, String, String, CommitSt, Game, History, String, Move) public void legalStateCirc(String, String, String, String, String, CommitSt, Game, History, Move) public void legalStateAction(String, String, String, String, String, CommitSt, Game, History, Move) public void legalStateConseq(String, String, String, String, String, CommitSt, Game, History, Move) public void legalStateLogCons(String, String, String, String, String, CommitSt, Game, History, Move) public void legalStatePurp(String, String, String, String, String, CommitSt, Game, History, Move) public void legalDenyCirc(String, String, String, String, String, CommitSt, Game, History, Move) public void legalDenyConseq(String, String, String, String, String, CommitSt, Game, History, Move) public void legalDenyLogCons(String, String, String, String, String, CommitSt, Game, History, Move) public void legalDenyPurp(String, String, String, String, String, CommitSt, Game, History, Move) public void legalDenyInitCircExist(String, String, String, String, String, CommitSt, Game, History, Move) public void legalDenyActExist(String, String, String, String, String, CommitSt, Game, History, Move) public void legalDenyNewStateExist(String, String, String, String, String, CommitSt, Game, History, Move) public void legalDenyGoalExist(String, String, String, String, String, CommitSt, Game, History, Move) public void legalDenyValueExist(String, String, String, String, String, CommitSt, Game, History, Move) public void legalAskCirc(String, String, String, String, String, CommitSt, Game, History, Move) public void legalAskAct(String, String, String, String, String, CommitSt, Game, History, Move) public void legalAskConseq(String, String, String, String, String, CommitSt, Game, History, Move) public void legalAskLogCons(String, String, String, String, String, CommitSt, Game, History, Move) public void legalAskPur(String, String, String, String, String, CommitSt, Game, History, Move) public String p1CheckDenial(String, String, String, String) public String p2CheckDenial(String, String, String, String) public void printComStores(String, String)

2.5 Play Class

Play
public static BufferedReader keyboardInput
public Play() public static void main(String[] args)

2.6 String Class

String
public boolean equals(String) public int compareTo(String) public String substring(int,int) public int indexOf(int) public String concat(String)

2.7 BufferedReader Class

BufferedReader
public String readLine()

2.8 FileWriter Class

FileWriter
public void close() public void write(String)

3. Design

Presented below are tables giving a summary of all fields, constructors and methods used in each class of the implementation. Each table explains the purpose and interaction of all the elements in the individual classes. The tables all follow the format used by Sun to describe the Java API, which can be found at: http://www.java.sun.com/reference/api/index.html

3.1 History Class

Field Summary

private String[][]	History A private instance field which is a 2D array to store the history of the game containing info about who made the move, the locution name, the legal status of the commitment and its content.

Constructor Summary

History() Constructs an instance of the class History.

Method Summary

public void	updateHistory(String speaker, String move) Method to update the first three elements of the history array and takes the speaker and the move as arguments.
public void	legalUpdate(String contents) Method to update certain elements of the history once a move has been proved to be legal and takes the content of the move as an argument.
public void	illegalUpdate(String contents) Method to update certain elements of the history once a move has been proved to be illegal and takes the content of the move as an argument.
public void	printHistory() Method to print the history array on screen and to file once the game has ended.

3.2 Play Class

Field Summary

public static BufferedReader	keyboardInput An class instance to facilitate input from the input stream

Constructor Summary

Play() Constructs an instance of the class Play.

Method Summary

public static void	main(String[] args) Main method to get the names of the two players, assign these to a speaker and hearer role, create new instances of Game, CommitSt, History, Move and call the firstMove method to get the speaker's first move.

3.3 Move Class

Field Summary

private Boolean	movePossible A private instance field to hold the boolean variable to say whether a move is possible or not.
private String[]	possibleMoves An array of Strings to hold all the locutions that it is possible to utter in this game.
private static final int	LOWER_BOUND A class constant to hold the lower bound of the array index
private static final int	ARRAY_LENGTH A class constant to hold the length of the possibleMoves[] array

Constructor Summary

Move() Constructs an instance of the class Move.

Method Summary

public void	checkPossible(String move, String speaker, String hearer, String play1, String play2, CommitSt newComSt, Game newGame, History newHist, Move newMove) Method to check that the move that has been chosen is a possible move in this game and thus is in the possibleMoves[] array.
public void	successful(String move, String speaker, String hearer, String player1, String player2, CommitSt comSt, Game game, History histo, Move moveInst) Method using selection statements to call the appropriate method to check the legality of the move, depending upon which move has been chosen.

3.4 Game Class

Field Summary

private String	speaker A private instance field to hold the name of the player assigned to the speaker role.
private String	hearer A private instance field to hold the name of the player assigned to the hearer role.
private String[]	possibleMoves An array of Strings to hold all the locutions that it is possible to utter in this game.
public static BufferedReader	keyboardInput An class instance to facilitate input from the input stream

Constructor Summary

Game() Constructs an instance of the class Game.

Method Summary

public void	firstMove(String player1, String player2, CommitSt cs, Game dg, History his, Move mov) Method, used only once in each game, to assign players to speaker and hearer roles and read in the first move.
public void	turnFinished(String hearer, String speaker, String play1, String play2, CommitSt comS, Game gme, History hi, Move mo) Method to check if the current speaker has finished their turn or not and if they have finished then swap the speaker and hearer over.
public void	makeMove(String speaker, String hearer, String player1, String player2, CommitSt comStore, Game theGame, History theHist, Move theMove) Method to read in the speaker's chosen move which in turn calls the checkPossible method to check if the move read in is a possible move in this game.

3.5 CommitSt Class

Field Summary

private String[][]	play1ComSt A 2D array to hold player 1's commitment store and it consists of a locution name, status of commitment and content of commitment.
private String[][]	play2ComSt A 2D array to hold player 2's commitment store and it consists of a locution name, status of commitment and content of commitment.
private String	content A String to hold the content of the speaker's move.
public static BufferedReader	keyboardInput An class instance to facilitate input from the input stream.

Constructor Summary

CommitSt() Constructs an instance of the class CommitSt.

Method Summary

public void	legalStateCirc(String move, String hearer, String speaker, String play1, String play2, CommitSt cs, Game dGame, History hist, Move mMove) Method to check the pre-conditions of the 'state circumstances' move. If they hold then update the history and the player's commitment store, print them on screen and call the turnFinished method to see if they have finished their turn. If the pre-conditions do not hold call the illegalMove method.
public void	legalStateAction(String move, String hearer, String speaker, String play1, String play2, CommitSt cs, Game dGame, History hist, Move mMove) Method to check the pre-conditions of the 'state action' move. If they hold then update the history and the player's commitment store, print them on screen and call the turnFinished method to see if they have finished their turn. If the pre-conditions do not hold call the illegalMove method.
public void	legalStateConseq(String move, String hearer, String speaker, String play1, String play2, CommitSt cs, Game dGame, History hist, Move

	mMove) Method to check the pre-conditions of the 'state consequences' move. If they hold then update the history and the player's commitment store, print them on screen and call the turnFinished method to see if they have finished their turn. If the pre-conditions do not hold call the illegalMove method.
public void	legalStateLogCons(String move, String hearer, String speaker, String play1, String play2, CommitSt cs, Game dGame, History hist, Move mMove) Method to check the pre-conditions of the 'state logical consequences' move. If they hold then update the history and the player's commitment store, print them on screen and call the turnFinished method to see if they have finished their turn. If the pre-conditions do not hold call the illegalMove method.
public void	legalStatePurp(String move, String hearer, String speaker, String play1, String play2, CommitSt cs, Game dGame, History hist, Move mMove) Method to check the pre-conditions of the 'state purpose' move. If they hold then update the history and the player's commitment store, print them on screen and call the turnFinished method to see if they have finished their turn. If the pre-conditions do not hold call the illegalMove method.
public void	legalDenyCirc(String move, String hearer, String speaker, String play1, String play2, CommitSt cs, Game dGame, History hist, Move mMove) Method to check the pre-conditions of the 'deny circumstances' move by calling the p1CheckDenial/p2CheckDenial method. If they hold then update the history, the player's commitment store to contain the denial made on one the opposing player's commitments and call the askAccept method to ask the opposing player if they accept the denial made. If the pre-conditions do not hold call the illegalMove method.
public void	legalDenyConseq(String move, String hearer, String speaker, String play1, String play2, CommitSt cs, Game dGame, History hist, Move mMove) Method to check the pre-conditions of the 'deny consequences' move by calling the p1CheckDenial/p2CheckDenial method. If they hold then update the history, the player's commitment store to contain the denial made on one the opposing player's commitments and call the askAccept method to ask the opposing player if they accept the denial made. If the pre-conditions do not hold call the illegalMove method.
public void	legalDenyLogCons(String move, String hearer, String speaker, String

	play1, String play2, CommitSt cs, Game dGame, History hist, Move mMove) Method to check the pre-conditions of the 'deny logical consequences' move by calling the p1CheckDenial/p2CheckDenial method. If they hold then update the history, the player's commitment store to contain the denial made on one the opposing player's commitments and call the askAccept method to ask the opposing player if they accept the denial made. If the pre-conditions do not hold call the illegalMove method.
public void	legalDenyPurp(String move, String hearer, String speaker, String play1, String play2, CommitSt cs, Game dGame, History hist, Move mMove) Method to check the pre-conditions of the 'deny purpose' move by calling the p1CheckDenial/p2CheckDenial method. If they hold then update the history, the player's commitment store to contain the denial made on one the opposing player's commitments and call the askAccept method to ask the opposing player if they accept the denial made. If the pre-conditions do not hold call the illegalMove method.
public void	legalDenyInitCircExist(String move, String hearer, String speaker, String play1, String play2, CommitSt cs, Game dGame, History hist, Move mMove) Method to check the pre-conditions of the 'deny initial circumstances exist' move by calling the p1CheckDenial/p2CheckDenial method. If they hold then update the history, the player's commitment store to contain the denial made on one the opposing player's commitments and call the askAccept method to ask the opposing player if they accept the denial made. If the pre-conditions do not hold call the illegalMove method.
public void	legalDenyActExist(String move, String hearer, String speaker, String play1, String play2, CommitSt cs, Game dGame, History hist, Move mMove) Method to check the pre-conditions of the 'deny action exists' move by calling the p1CheckDenial/p2CheckDenial method. If they hold then update the history, the player's commitment store to contain the denial made on one the opposing player's commitments and call the askAccept method to ask the opposing player if they accept the denial made. If the pre-conditions do not hold call the illegalMove method.
public void	legalDenyNewStateExist(String move, String hearer, String speaker, String play1, String play2, CommitSt cs, Game dGame, History hist, Move mMove) Method to check the pre-conditions of the 'deny resultant state exists' move by calling the p1CheckDenial/p2CheckDenial method.

	If they hold then update the history, the player's commitment store to contain the denial made on one the opposing player's commitments and call the askAccept method to ask the opposing player if they accept the denial made. If the pre-conditions do not hold call the illegalMove method.
public void	legalDenyGoalExist(String move, String hearer, String speaker, String play1, String play2, CommitSt cs, Game dGame, History hist, Move mMove) Method to check the pre-conditions of the 'deny goal exists' move by calling the p1CheckDenial/p2CheckDenial method. If they hold then update the history, the player's commitment store to contain the denial made on one the opposing player's commitments and call the askAccept method to ask the opposing player if they accept the denial made. If the pre-conditions do not hold call the illegalMove method.
public void	legalDenyValueExist(String move, String hearer, String speaker, String play1, String play2, CommitSt cs, Game dGame, History hist, Move mMove) Method to check the pre-conditions of the 'deny value exists' move by calling the p1CheckDenial/p2CheckDenial method. If they hold then update the history, the player's commitment store to contain the denial made on one the opposing player's commitments and call the askAccept method to ask the opposing player if they accept the denial made. If the pre-conditions do not hold call the illegalMove method.
public void	legalAskCirc(String move, String hearer, String speaker, String play1, String play2, CommitSt cs, Game dGame, History hist, Move mMove) Method to check the pre-conditions of the 'ask circumstances' move by asking the player to enter the topic which they are inquiring about and checking that they do not already have commitments on this topic. If these pre-conditions hold then update the history and call the answerAsk method to ask the opposing player to respond to the request for information. If the pre-conditions do not hold call the illegalMove method.
public void	legalAskAct(String move, String hearer, String speaker, String play1, String play2, CommitSt cs, Game dGame, History hist, Move mMove) Method to check the pre-conditions of the 'ask action' move by asking the player to enter the topic which they are inquiring about and checking that they do not already have commitments on this topic. If these pre-conditions hold then update the history and call the answerAsk method to ask the opposing player to respond to the request for information. If the pre-conditions do not hold call the illegalMove method.

public void	legalAskConseq(String move, String hearer, String speaker, String play1, String play2, CommitSt cs, Game dGame, History hist, Move mMove) Method to check the pre-conditions of the 'ask consequences' move by asking the player to enter the topic which they are inquiring about and checking that they do not already have commitments on this topic. If these pre-conditions hold then update the history and call the answerAsk method to ask the opposing player to respond to the request for information. If the pre-conditions do not hold call the illegalMove method.
public void	legalAskLogCons(String move, String hearer, String speaker, String play1, String play2, CommitSt cs, Game dGame, History hist, Move mMove) Method to check the pre-conditions of the 'ask logical consequences' move by asking the player to enter the topic which they are inquiring about and checking that they do not already have commitments on this topic. If these pre-conditions hold then update the history and call the answerAsk method to ask the opposing player to respond to the request for information. If the pre-conditions do not hold call the illegalMove method.
public void	legalAskPur(String move, String hearer, String speaker, String play1, String play2, CommitSt cs, Game dGame, History hist, Move mMove) Method to check the pre-conditions of the 'ask purpose' move by asking the player to enter the topic which they are inquiring about and checking that they do not already have commitments on this topic. If these pre-conditions hold then update the history and call the answerAsk method to ask the opposing player to respond to the request for information. If the pre-conditions do not hold call the illegalMove method.
public void	illegalMove(String speaker, String hearer, String play1, String play2, CommitSt cs, Game dGame, History hist, String move, Move mMove) Method to inform the players when they have tried to make an illegal move. Updates the history to show an illegal move has been made and by who, informs the player that they have made an illegal move and calls the makeMove method to ask the user to choose a legal move.
public void	answerAsk(String moveMade, String hear, String speak, String p1, String p2, CommitSt comSto, Game theGame, History his, String contents, String rep, Move mMove) Method to let the speaker reply to the 'ask' move just previously

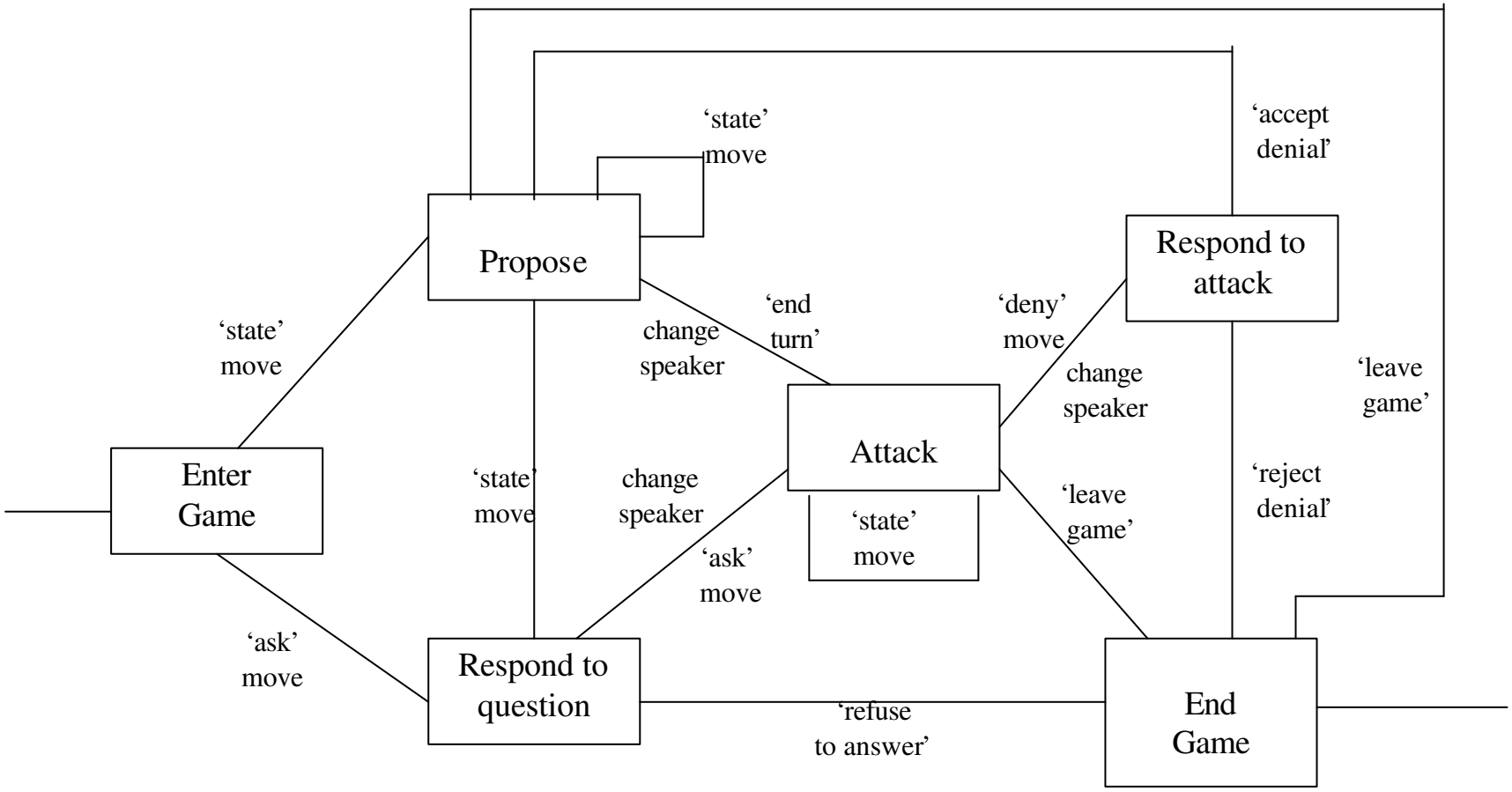
	made by the other player. If they reply with a 'state' move call the successful move to check the move is legal. If they reply with 'don't know' update their commitment store to state that there's no commitment made on this point then ask them if they've finished their turn. If they choose 'exit' inform them that the game has ended and call the methods to print the history and commitment stores to file and to the screen then exit the game. If the player doesn't choose a valid response ask them to re-enter their choice and choose a valid response.
public void	askAccept(String moveMade, String hear, String speak, String p1, String p2, CommitSt comSto, Game theGame, History his, String contents, Move mMove) Method to ask the speaker whether or not they accept the denial just made on an element of their position. If they accept the denial then call the legalAccept method to check the legality of this move. If they reject the denial then inform them that disagreement has been reached and therefore the game cannot continue and call the methods to print the history and commitment stores to file and to the screen then exit the game. If the player doesn't choose a valid response ask them to re-enter their choice and choose a valid option.
public void	legalAccept(String move, String hearer, String speaker, String play1, String play2, CommitSt cs, Game dGame, History hist, String content, Move mMove) Method to check the pre-conditions of the 'accept denial' move. If the pre-conditions hold update the history and commitment stores then ask the player if they have finished their turn. If the pre-conditions do not hold call the illegalMove method.
public void	printComStores(String player1, String player2) Method to print the player's commitment stores on screen and to file once the game has ended.
public String	p1CheckDenial(String move, String against, String speaker, String hearer) Method to let player 1 enter the content of an attack and check that player 2's commitment already contains this content, when player 1 is making a 'deny' move. If the content's do not match ask player 1 to re-enter the content and ensure it is something player 2 is already committed to. If the contents do match return this as a String.
public String	p2CheckDenial(String move, String against, String speaker, String hearer) Method to let player 2 enter the content of an attack and check that player 1's commitment already contains this content, when player 2 is making a 'deny' move. If the content's do not match ask player 2 to re-enter the content and ensure it is something player 1 is already committed to. If the contents do match return this as a String.

Summary tables for the pre-defined String, FileWriter and BufferedReader classes can been found in the Java API documentation at:
http://www.java.sun.com/reference/api/index.html

5. State Transition Diagram

Figure 2 below shows a simple state transition diagram for the protocol. It shows the types of moves that the players can make and the choice of move which is then available in the new state. It also shows the moves that lead to the roles of speaker and hearer being switched and how the game can terminate. The diagram does not show the specific details of all moves that can be made, only the types of moves e.g. 'state', 'deny', 'ask', etc.

Figure 2 - State Transition Diagram for PARMA Dialogue Game Protocol



6. Evaluation and Discussion

6.1 General Evaluation

The implementation of the dialogue game specified above has successfully been completed. The completed code allows two human players to play the game, in accordance with the given specification. However, there are a number of issues which came to light through implementing the dialogue game. Many issues which were encountered throughout the implementation forced us to re-evaluate the design to incorporate changes which we thought were necessary. There were also many issues which we encountered but chose not to incorporate into the design or to implement. There are a number of reasons for this; first and foremost there are more possible implementations of the game than we first anticipated and we felt that a well defined application context was needed to motivate the choice of possibilities to realise; secondly, we encountered a number of issues that require further theoretical discussion, as there are a number of possible solutions to these issues and currently we are unsure about which solutions are the most appropriate.

The rest of this evaluation section is divided up into points that we see as falling into one of five categories, namely; General Insights, Specific Specification Errors, Under Specification/Unanticipated Possibilities, Implementation Simplifications and an Additional Point for Discussion, all of which are described in detail below.

General Insights

6.2 Pre-Conditions and Post-Conditions

One of the most fundamental differences which immediately came to light when starting to implement the dialogue game was the difference between the original specification of the pre and post-conditions in the design and how we actually implemented them.

The major difference which occurred in all the 'state' locutions was the pre-condition of set membership. In the design the pre-conditions contained the rule that when committing to some element of a position, the speaker must be committed to the fact that the element exists in the set of possible elements. However, when implementing the game it became apparent that this should not be a pre-condition, but actually a post condition of making one of the 'state' locutions. For example, when making the 'state circumstances (R)' act the specification stipulated that in order for this act to be legally made the speaker must already be committed to the fact that R exists in the set of possible circumstances. However, when implementing this we realised that this should be changed from a pre-condition to a post-condition in all the 'state' locutions. This means that when the speaker makes the 'states circumstances' move, if it is legal, then their commitment store is updated to contain the fact that they implicitly commit to R existing in the set of possible circumstances, as well as being committed to the propositional content of 'state circumstances (R)'. Commitments to elements existing in the set of possible elements are never explicitly made by the speaker in the course of a game, as they are only ever added implicitly to the commitment stores when a

speaker makes a 'state' act. Such 'set membership' commitments can however be explicitly attacked, as was set out in the original specification and design of the dialogue game.

The reason that this issue arose was due to there being a difference in perspective. Commitment to a proposition is a pre-condition for sincerely uttering it, so the player will see this as a pre-condition, and this is what we specified in the original specification. Uttering the proposition allows observers (assuming sincerity) to infer commitment, so as far as the public commitment store is concerned this is a post-condition. The dialogue game protocol is conducted from the point of view of the referee so we altered the specification to reflect this.

6.3 Relationship Between Elements in a Player's Position

The implementation of the game does not explicitly tie together the elements of the players' positions. By this we mean that there are no data structures to hold all the individual elements (i.e. the R, A, S, G, V, D) of a player's position. Instead, the content of a move is associated with the type of move which commits the player to the content. For example, when making the 'state consequences (A,R,S)' move there are no data structures to hold data for the A, R, and S individually. Instead, the move has a natural language content associated with it which reveals what the A, R and S represent. The reason we chose to implement the content of moves in this way was to make it easier to use and more understandable for human players, as this approach is closer to natural language. However, if this game was to be automated we would have to change this representation to some appropriate logic in order for it to be useable by autonomous agents.

The above point also raises another matter relating to how the attacks are used by the players. The code does not check associations between elements of a position and instead it leaves this to the participants. For example, a player could legally have the following moves in their commitment store:

Move	Status	Content
state circumstances	1	(It is raining outside)
circ exist	1	(It is raining outside exists in the set of possible circumstances)
state action	1	(Take an umbrella if you go out)
act exist	1	(Take an umbrella exists in the set of possible actions)
state consequences	1	(The price of bread will rise)
state exist	1	(The price of bread will rise exists in the set of possible resultant states)

The above commitments will all be legally accepted in the game. However, it is obvious to see that the cost of bread has nothing to doing with it raining outside and taking an umbrella out. Therefore, it is the duty of the opposing player to point out this inconsistency in the form of an attack. We could have implemented the program

to disallow such unrelated topics to be discussed in this way but, that would be hard coding the attacks and we want the players to recognise inconsistencies so they will enforce the attacks themselves and not have the code imposing this.

The game is understood in syntactic terms and therefore the players must be responsible for the semantic content being relevant (as in natural dialogues), unless the referee were to be omniscient, which could be appropriate in some contexts. In order to aid the communication process we expect players of the game to follow sensible and helpful principles, such as the maxims set out by Grice in [6].

Specific Specification Errors

6.4 Denials on Set Membership

This point is related to the previous comments on attacking a player's 'set membership' commitments. When implementing the 'deny' locutions, we realised that we had omitted a post-condition of making these moves. The specification did not include anything to say that when a player successfully makes a denial on the existence of an element and the opposing player accepts this, the original commitment to that element should also be altered. For example, a player could have the following in their commitment store:

Move	Status	Content
state action	1	Let's catch the 2.30pm train
action exists	1	Catching the 2.30pm train exists in the set of possible actions

The opposing player can make the 'deny action exists' attack to state that they believe that 'Catching the 2.30pm train' does not exist in the set of possible actions (because the time is now 2.45pm, for example). If the first player accepts this then their commitment store is updated as follows:

Move	Status	Content
accept non existence	-1	Let's catch the 2.30pm train
accept denial	-1	Catching the 2.30pm train exists in the set of possible actions

When making such an attack, the original specification did not contain the post-condition of changing the 'state action' move to an acceptance of a denial, as well as changing the 'action exists' move to the acceptance of a denial. This is clearly needed because if a player is committed to the non existence of an element then they cannot be committed to a proposition about it. So, the denial of possibility entails the denial of truth.

6.5 Omission of an Attack in the Underlying General Theory of Persuasion

After implementing and testing the game it came to our attention that there seems to be an attack missing from the original theory upon which the game is based. Namely, we believe that **attack 1**: "R is not the case" has a variant which executes the attack with a different degree of force, as is the case for the other three main forms of attack in the theory. As well as having the attack "R is not the case", which we shall call **attack 1a,** we believe that the following attack should also be included in the theory:

attack 1b: "R is not the case and there is a circumstance Q ? States, where R ? Q, such that Q is the case".

The implementation brought this omission to light and the code now allows for this attack to be made in the game. We have also now added this attack to our general theory.

Under Specification/Unanticipated Possibilities

6.6 Next Available Moves

One considerable unforeseen problem that we encountered involved presenting the players with a list of the next available moves that they could legally make, once a new commitment has been accepted. In the specification for the game we identified a list of next available moves to accompany each locution that has been legally made. However, we now realise that we failed to take into consideration the fact that when playing the game the list is not as linear as we first thought. For example, when the speaker makes a 'state action (A_1)' move then the only move that this speaker can go on to make next is the 'state consequences (A_1,R_1,S_1)' move, according to the original specification of the game. However, this is not the case for a number of reasons.

Firstly, the speaker can go on to make a 'state action (A_2)' move or a 'state circumstances (R_2)' move, as the speaker may make multiple statements about individual elements within their position.

Secondly, if the speaker has already made the 'state consequences (A,R,S)', 'state logical consequences (S,G)' or the 'state purpose (G,V,D)' move previously in the game then they can legally repeat any of these moves (as long as the repeated moves contain different propositional content from the original moves).

Thirdly, the specification also documented the list of moves which are available to the attacker, should they wish to make an attack on the element of the position which has just been stated. We also found these lists to be incorrect when implementing the game. This is due to the fact that a speaker can make multiple statements about their position in one turn. For example, a speaker can make the 'state circumstances (R)' move, the 'state action (A)' move and the 'state consequences (A,R,S)' move all in one turn before the hearer has a chance to respond to the first element, i.e. the speaker's 'state circumstances (R)' move. This means that the list of moves which is

available to the hearer, once the speaker has finished their turn, must take into account the fact that all elements stated by the speaker can now be attacked.

All these points obviously show that the 'next available moves', detailed in the original specification, failed to take the above facts into account. We became aware of the above issues, with regards to the next available moves, whilst implementing the dialogue game. However, we decided against presenting the list to the users in the implementation, as it would offer them little help due the fact that the list would be so large. Natural dialogue contains a great degree of flexibility and if the same degree of flexibility is allowed in a dialogue game, then the choice of moves is so extensive that little support can be given. Particular games should therefore limit choice, so that the user is forced to make a sensible move, even at the cost of disallowing some perfectly natural moves. For now, we just note that the next available moves are entirely defined by the pre-conditions of the individual locutions.

6.7 Repeated Statement of Attacked Commitments

The previous point of accepting denials on commitments raised another issue which we had not previously accounted for in the design for the game, namely disallowing a player to repeatedly commit to some specific proposition after having previously accepted an attack on that commitment, which resulted in the player being committed to the negation of the original commitment. The design did not specifically prohibit this which means that cycles could occur in the dialogue, leading to infinite repetition of locutions, which would result in the dialogue being interminable. This is obviously a scenario which is undesirable. But, if a natural dialogue is allowed to take place then the repetition of statements also needs to be permitted, as in natural dialogue people do often make such unhelpful moves. However, we do believe that this should be disallowed in a computer controlled game and therefore it would be desirable to include termination rules in our protocol to ensure that repeated statements are not infinitely made.

6.8 'Ask' Locutions

Implementing the 'ask' locutions proved to be quite a difficult task due to the fact that we now believe that the specification of the pre-conditions for all the 'ask' moves was incorrect, with respect to the amount of freedom we wished to give the players with these moves. The original specification stated that the pre-conditions of all the 'ask' moves were that the speaker making the 'ask' move was not already committed to some circumstances. However, it was decided that even if a speaker is already committed to a particular element of a position they should be allowed to go on and ask about that element (for example, "what are the consequences of breaking the law?") in the future course of the game, as long as the propositional content they are enquiring about does not already exist as a commitment in their commitment store (for example, they don't already have some commitment in their commitment store which relates to their views on the consequences of breaking the law). This issue came to light as we realised that it is possible to tell the truth, but not the whole truth, even if this retention of information is not intended. Originally we saw people as

making complete statements. This, however, is not natural. Moreover, it is the audience rather than the speaker who should judge relevance. Therefore, we have changed both the design and the implementation to incorporate these altered pre-conditions to allow for the possibility of asking for additional information. We also implemented a change in control of the dialogue immediately after a legal 'ask' move has been made in order to force the hearer to immediately respond to the 'ask', with the possible responses being either; the statement of the particular element, a 'don't know' response, or the option to leave the game.

6.9 'Deny' Moves in Commitment Stores

The specification of the dialogue game did not include the insertion of 'deny' moves into the commitment stores of the players. However, we did include this feature in the implementation. During the specification stage we thought it only necessary to include the 'deny' moves in the history of the dialogue but, after reconsidering this point we thought it would also be necessary to include them in the player's commitment stores. The reason for this is that in denying an element of a position, a player is making a commitment to the negation of an element of their opponent's position. This is very different from committing outright to the negation of the proposition, without making a 'deny' move, but making a 'state' move instead. For example, player 1 could have the following in their commitment store:

Move	Status	Content
state circumstances	1	It is raining outside

which reads: "I am committed to the circumstance 'It is raining outside'".

Player 2 could then make a 'deny circumstances' attack on the above commitment and if it is legal then player 2's commitment store will be updated as follows:

Move	Status	Content
deny circumstances	1	It is raining outside

which reads: "I am committed to denying the circumstance 'It is raining outside'".

This is very different from stating the circumstance 'It is not raining outside', which would make the player's commitment store as follows:

Move	Status	Content
state circumstances	1	It is not raining outside

There is an obvious difference between the above two statements and we believe it to be important to make this distinction obvious in the commitment stores. It becomes an even more important point when classifying the game in terms of the attacks listed in our general theory of persuasion. According to the game's specification, the attacks are made up of mixtures of 'deny' moves and 'state' moves so it is important to see when inspecting the player's commitment stores that it is obvious which moves were actually denials on the opposing player's position and not just the statement of the opposite of the opponent's commitment.

The above point also brings to light another difficulty regarding semantics in the implementation. If a player makes a commitment to some proposition and the opposing player disagrees with this commitment they should then make a 'deny' move. However, there is nothing in the implementation to stop the opposing player just stating the opposite, rather than making a denial. This is recording the difference between a volunteered denial and a denial in response to a challenge. To illustrate this, again using the above example, player 1 might have the following commitment in their commitment store:

Move	Status	Content
state circumstances	1	It is raining outside

Player 2 may disagree with this and therefore they should make the 'deny circumstances' move with 'It is raining' as the content. However, there is nothing in the code to stop player 2 just stating the opposite of player 1 i.e. making the 'state circumstances' move with the content 'It is not raining'. This obviously poses a problem when analysing what is and what isn't an attack. The problem has arisen due to the fact that this game (as well as many other dialogue games) attaches labels to statements made by the players, whereas in natural language we usually recognise when someone is making an attack on our views without having to explicitly state what they are doing. We are therefore relying on the goodwill of the players to choose the appropriate moves in accordance with how the rules and incentives of the protocol work.

6.10 Context Dependence

The points made in the above section also led us to believe that the game is more context dependant than we first thought. The language that is used by two people having a conversation varies greatly depending upon the situation, the topic of discussion and the relationships between the players. For example, in a court of law statements are usually explicitly and fully stated to try and eliminate the possibility of ambiguity and attacks on positions are likely to be more explicit too. Conversely, when two people are having an everyday conversation about a trivial topic, such as the weather, then they tend to be more ambiguous and use less explicit language. Again, this is concerned with the flexibility of natural dialogue being opposed to the ability to infer things about the dialogue. As mentioned in the above subsection, the game does rely on the goodwill of the players to stick to the rules of the protocol, as

not all rules are explicitly checked by the program. This would obviously make the program unsuitable for use in a domain such as the legal one, where players cannot be relied upon to adhere to the rules.

We have specified two different versions of the game; a loose game and a strict game. We have implemented the strict version and this still relies upon the users' goodwill, to a small extent, to ensure that the game proceeds accordingly. This has led us to realise that it may be necessary to have even more strict pre-conditions for certain moves if the game was to be used in a necessarily strict domain, such as the legal one. The best choice of restrictions on unfettered choice needs to be made against consideration of the context. Different contexts will urge different choices.

Implementation Simplifications

6.11 Retraction of Commitments

Before the implementation of the game commenced, we were not entirely sure how we were going to deal with the concept of retracting commitments. In the specification we proposed to have a 'retract' locution to enable the participants to retract commitments that had been defeated by an attack. However, when coding the 'deny' locutions we decided not to include the 'retract' locution. We dealt with this issue in a different manner in order to allow the players' commitment stores to display more descriptive information about the acceptance of denials on a particular element of a player's position. To do this we had to hard code stricter control of the moves that can be made when a player makes a 'deny' locution. To clarify, when a player legally makes a 'deny' locution, control is immediately passed to the other player to force them to respond to the attack. At this point they must either accept the denial or reject it, and if they reject it then the game terminates with conflict on that point. If the denial is accepted then the player's commitment is not taken out of their commitment store but it is overwritten. This involves changing the status of the commitment from a 1 (which indicates that a player is committed to a proposition) to a -1 (which indicates that a player is committed to the negation of a proposition), as well as changing the name of the move that brought about the commitment to the proposition in question. For example, player 1 could have the following commitment in their commitment store:

Move	Status	Content
state circumstances	1	It is raining outside

which reads: "I am committed to the circumstance 'It is raining outside'".

Player 2 could then make a 'deny circumstances' attack on the above commitment and if this is accepted by player 1, then player 1's commitment store will be updated as follows:

Move	Status	Content
accept denial	-1	It is raining outside

which reads: “I accept the denial made upon this state circumstances move and I am now committed to the negation of the circumstance ‘It is raining outside’”.

So, by altering a commitment’s status and name we can see which commitments have been challenged and accepted. This eliminates the need for retraction and also gives us more descriptive commitment stores, which may in turn be useful in future work examining strategies that players could use to persuade the opposition into accepting an attack.

If the status of a commitment is left hanging, then choice proliferates. Dialogue games which require explicit change of focus do become rather complicated, as can be seen in [3], compared to the useful simplification of games such as Two Party Immediate Response disputes, as detailed in [4]. We chose to simplify matters in our protocol by demanding an immediate resolution of the status of a proposition under challenge and this eliminates the need for a focusing mechanism.

Additional Point of Discussion

6.12 An Alternative Implementation

After reflecting on some of the issues previously raised in this section, we have concluded that the implementation considered here poses many problems for casual users of the system. In order to correctly follow the protocol the users must have prior knowledge of the underlying theory of persuasion. If they do not have prior knowledge of the theory then they will be unable to recognise which locutions need to be chosen in order to realise the correct attack, in a given situation. The users must also be familiar with the names and meanings of the locutions used to represent the statement and denial of a position. As well as these usability problems, we mentioned in the previous section that the dialogue game does rely somewhat on the goodwill of the players to follow the protocol, as it is not always strictly enforced by the actual program.

Some of these problems have arisen due to the amount of freedom of expression afforded by the program and this leaves the users with an overwhelming variety of options to select between. All these points related to problems with the usability of the program are obviously undesirable. Therefore, we have addressed these issues by going on to implement our theory of persuasion in an entirely different format.

We have developed an online discussion forum, named PARMENIDES (Persuasive ArguMENt In DEmocracieS) which allows a much simpler form of interaction to take place. The user is guided through a series of web pages in order to elicit their views on a particular topic, in accordance with our theory. The user interaction occurs through a simple web based interface which guides them in a structured fashion through a justification of an action, giving opportunities to disagree at selected points.

Each of these disagreements represents one of the attacks from our theory of persuasion, so the exact nature of the disagreement can be unambiguously identified. By constraining the choice of the user in such a way, the need for them to understand the underlying argumentation scheme and thus select the correct moves is removed. The users' responses are written to a database so we are able to gather and analyse the information in order to identify what points of the argument are more strongly supported than others.

This system has been successfully implemented and we are satisfied that it overcomes many of the usability problems presented by the Java program, which have been highlighted in this section. Details of the PARMENIDES online discussion forum can be found in [2].

7. Conclusions

This report has presented an implementation of the PARMA protocol, a dialogue game protocol previously proposed by the authors. Implementing the dialogue game has proved to be a very useful task as we have shown that our general theory of persuasion can be conducted via computer mediated dialogues of this form. There are still some alterations to be made to the code to improve it but, to date all the important elements of the underlying theory are included in the implementation.

This implementation has also raised a number of interesting issues in relation to our underlying argumentation scheme, as well as leading us to what we believe is an improved alternative implementation, in the form of the PARMENIDES system detailed above. We now intend to focus on this system to extend our theory and implementation further. We hope to include other elements, such as counter attacks (which have not yet been explored) and allow the construction of positive alternative arguments, as the system currently focuses on the negative criticism of arguments.

To conclude we summarise the three main general insights which have arisen through our evaluation of the implemented dialogue game protocol:

1) The referee cannot use pre-conditions based on mental states of the participants: he infers these from the moves the players make.
2) Natural dialogue is very flexible. Giving support requires constraints and what constraints are appropriate depends on context and purpose.
3) Goodwill and some co-operation is required to make sensible progress and this is again due to the fact that natural dialogue is so flexible.

References

[1] K.M. Atkinson, T.J.M Bench-Capon and P.J. McBurney (2004). A Dialogue Game Protocol for Multi-Agent Argument over Proposals for Action. In submission.

[2] K.M. Atkinson, T.J.M Bench-Capon and P.J McBurney (2004). PARMENIDES: Facilitating Democratic Debate. In Proc. *Third Intern. Conf. eGovernment (EGOV-2004)*, Zaragoza, Spain. Springer, LNCS, Berlin. *To appear*.

[3] T.J.M. Bench-Capon, T. Geldard, and P.H. Leng (2000), A method for the computational modelling of dialectical argument with dialogue games. *Artificial Intelligence and Law*, Vol 8, pp 233-254.

[4] P.E. Dunne and T.J.M. Bench-Capon (2003). Two party immediate response disputes: Properties and Efficiency. *Artificial Intelligence,* Vol 149 No 2 pp 221-50.

[5] K.M Greenwood and T.J.M. Bench-Capon and P.J. McBurney (2003). Towards a computational account of persuasion in law. In Proc. *Ninth Intern. Conf. AI and Law (ICAIL-2003)*, 22-31, ACM Press: New York, NY, USA.

[6] H. Grice (1975). Logic and Conversation. In P.Cole and J.L. Morgan, editors, *Syntax and Semantics III: Speech Acts*, pp41-58. Academic Press, New York City, NY, USA. Originally presented as part of the William James Lectures at Harvard University in 1967.