Exploring Agent-based Argumentation Dialogue Games

Daniel Bryant

Submitted for the Degree of
Master of Science in Internet Computing
from the
University of Surrey

Department of Computing
School of Electronics and Physical Sciences
University of Surrey
Guildford, Surrey, GU2 7XH, UK

August 2005

Supervised by: Prof. Paul Krause

I. Abstract

This dissertation explores the use of argumentation within agent-based communication and coordination, and more specifically, dialogue game-based scenarios where two or more agents (with potentially different interests) are attempting to reach a predefined goal through discourse. Over the past several years an increasing number of theoretical argumentation-based dialogue semantics have been developed and presented for various dialogue types, but their applicability has yet to be demonstrated in a functional agent-based application. Accordingly, this dissertation transforms the semantics of two dialogue types into a programmatic representation and presents a set of generic components that is capable of enforcing both the current programmatic representations produced and future representations. A fully functional agent-based dialogue application is also presented and demonstrates the enforcement of dialogue game rules (encapsulated in the semantic programmatic representation), preventing a participating agent from making an inappropriate move, or utterance, within the game and facilitating the expedient resolution of the dialogue.

"In the long history of humankind (and animal kind, too) those who learned to collaborate and improvise most effectively have prevailed"

Charles Darwin, 1809 – 1882.

II. Acknowledgements

Firstly, I would like to greatly thank my dissertation supervisor, Prof. Paul Krause, for all the guidance and opportunities he has provided over the past year. I would also like to greatly thank Dr. Lilian Tang, who has provided constant encouragement and support during this last year and also my previous undergraduate years. The support of these two people has made the transition from undergraduate study to research-based activities an interesting and fun experience. There are many friends and colleagues within the department of computing, too numerous to mention individually, who have also provided support and encouragement during this very intensive, yet very rewarding MSc course. To these people I also offer my gratitude and hopefully I have also helped you in someway. Finally, I would like to convey my deepest thanks to my parents and family who have continually supported the decisions I have made throughout my life, and have always listened and offered impartial advice when I needed it.

III. Table of Contents

I. ABSTRACT 2

II. ACKNOWLEDGEMENTS 3

III. TABLE OF CONTENTS 4

1 INTRODUCTION 8

1.1 OVERVIEW 8
1.2 AIMS OF THIS DISSERTATION 9
1.3 DISSERTATION STRUCTURE 11

2 LITERATURE REVIEW 12

2.1 INTRODUCTION 12
2.2 AGENT TECHNOLOGY OVERVIEW 12
2.2.1 Trends Leading to the Emergence of Agents 12
2.2.2 The Agent Based Paradigm 13
2.2.3 Agent Coordination Models 14
2.2.4 Agent-based Decision Models 17
2.3 CONTEMPORARY ARGUMENTATION WITHIN DIALOGUE 18
2.3.1 The Evolution of Argumentation 18
2.3.2 Dialogue Systems and Games 18
2.3.3 Elements of a Dialogue System 19
2.4 THE PURPOSE OF SEMANTICS WITHIN DIALOGUE 20
2.5 SUMMARY 20

3 COMPARISON OF DEPLOYMENT PLATFORMS 21

3.1 INTRODUCTION 21
3.2 CURRENT ARCHITECTURES AND STANDARDS 21
3.2.1 FIPA 22
3.2.2 MASIF 22
3.2.3 Summary of Standards Reviewed 23
3.3 REVIEW OF DEPLOYMENT PLATFORMS 23
3.3.1 Java – The De facto Agent Technology? 23
3.3.2 Java Remote Method Invocation (RMI) 24

3.3.3 Aglets Toolkit *25*
3.3.4 JavaSpaces *26*
3.3.5 Linda-like Variants *26*
3.4 SUMMARY 27

4 CREATING THE PROGRAMMATIC REPRESENTATION 29

4.1 INTRODUCTION 29
4.2 E-COMMERCE DIALOGUE OVERVIEW 29
4.3 SYNTAX AND SEMANTICS OF THE E-COMMERCE DIALOGUE 30
4.4 REPRESENTING THE SEMANTICS 31
4.4.1 Identifying the Required Information for Semantic Enforcement *31*
4.4.2 Designing the Utterance Class *32*
4.5 THE CONCEPT OF AN ENFORCEMENT MECHANISM 33
4.5.1 Designing the Semantic Policy *34*
4.6 ADAPTING THE DESIGN BY CONTRACT METHODOLOGY 34
4.6.1 The Assert Mechanism *34*
4.6.2 Adapting the Assert Mechanism *36*
4.6.3 Storing the Previous Utterances *36*
4.6.4 Documenting the Preconditions and Post-conditions *37*
4.7 THE SEMANTIC INTERFACE 37
4.8 SUMMARY 38

5 THE E-COMMERCE DIALOGUE PROTOTYPE 39

5.1 INTRODUCTION 39
5.2 PROTOTYPE OVERVIEW 39
5.3 ENFORCING THE SEMANTICS 40
5.3.1 Server-side Deployment *40*
5.3.2 Client-side Deployment *41*
5.3.3 Choosing the Deployment Location *42*
5.4 THE ARGUEAPPLET 42
5.4.1 Component Overview *42*
5.4.2 Making an Utterance *43*
5.4.3 Integrating the Utterance Class *44*
5.4.4 Enforcing the Semantic Policy *44*
5.4.5 An Example of the Enforcement Process *46*
5.4.6 Implementing the Communication Mechanism *46*
5.5 THE RMIGUARDIANAGENT SERVER 47
5.5.1 Component Overview *47*
5.5.2 The Guardian Agent Implementation *47*

5.6 LOGGING UTTERANCES 49
5.7 STORING UTTERANCES IN PERSISTANT STORAGE 49
5.8 TESTING 50
5.8.1 An Example Dialogue 50
5.8.2 Attempting to make an Inappropriate Utterance 52
5.8.3 Outstanding Issues 52
5.9 SUMMARY 53

6 THE GENERIC ENFORCEMENT COMPONENTS 54

6.1 INTRODUCTION 54
6.2 THE DELIBERATION DIALOGUE OVERVIEW 54
6.3 SYNTAX AND SEMANTICS OF THE DELIBERATION DIALOGUE 55
6.4 CHANGES NEEDED TO THE ENFORCEMENT MECHANISM 57
6.4.1 Modifying the Semantic Enforcement Method Parameters 57
6.4.2 Modifying the Checking of Previous Utterance Types 58
6.4.3 Checking Utterance Contents 60
6.4.4 Commitment Stores 61
6.5 CREATING THE GENERIC SEMANTIC ENFORCEMENT COMPONENTS 62
6.5.1 Overview 62
6.5.2 Designing the Dialogue Controller (Semantic Enforcement Mechanism) 62
6.5.3 Utilising the Template Pattern to Create a Generic Controller 63
6.5.4 Overview of DialogueController Slot Methods 64
6.5.5 Overview of the DialogueController Hook methods 65
6.6 RE-IMPLEMENTING THE E-COMMERCE SEMANTICS 65
6.6.1 Overview 65
6.6.2 Annotated Dialogue Controller UML Class Diagram 66
6.7 IMPLEMENTING THE DELIBERATION DIALOGUE SEMANTICS 67
6.7.1 Creating the Semantics Interface 67
6.7.2 Implementing the Dialogical Commitments 68
6.7.3 A Complex Utterance – The Retract Locution 69
6.8 SUMMARY 70

7 THE COMPLETE DIALOGUE APPLICATION 71

7.1 INTRODUCTION 71
7.2 NEW COMPONENTS OVERVIEW 71
7.3 REFACTORING THE USER INTERFACE CLASS (ARGUEAPPLET) 73
7.3.1 Composing an Agent Proxy 73
7.3.2 Notifying the Client of New Utterances 74
7.3.3 Generating the Utterance Type Buttons 75

7.3.4 Displaying the Current Commitment Store ... *75*
7.3.5 Modifying the Utterance Class ... *76*
7.4 DESIGNING A CONCRETE AGENT PROXY ... 76
7.4.1 Encapsulating the Communication Method ... *76*
7.4.2 Encapsulating the Agent State ... *78*
7.5 INCORPORATING THE DIALOGUE CONTROLLER ... 78
7.6 SUMMARY OF INTERACTION BETWEEN COMPONENTS ... 79
7.6.1 Overview ... *79*
7.6.2 Outbound Communication ... *79*
7.6.3 Inbound Communication ... *80*
7.7 SERVER SIDE COMPONENTS ... 81
7.8 TESTING ... 81
7.8.1 An Example Dialogue ... *82*
7.8.2 Attempting to Make a Malformed Utterance ... *84*
7.8.3 Attempting to Make an Inappropriate Utterance ... *84*
7.9 SUMMARY ... 85

8 DISCUSSION ... 86

8.1 INTRODUCTION ... 86
8.2 CLIENT-SIDE COMPONENTS ... 86
8.3 GENERIC ENFORCEMENT COMPONENTS ... 87
8.3.1 Overview ... *87*
8.3.2 Supporting the Deliberation Dialogue ... *88*
8.4 SERVER-SIDE COMPONENTS ... 89
8.5 COMPLIANCE WITH STANDARDS ... 90
8.6 RELATED WORK ... 91
8.7 SUMMARY ... 91

9 CONCLUSIONS AND FUTURE WORK ... 92

9.1 OVERVIEW ... 92
9.2 CONCLUSIONS AND FUTURE WORK ... 92
9.3 FINAL SUMMARY ... 94

10 REFERENCES ... 95

11 APPENDIX I - PROTOTYPE APPLICATION UML DIAGRAMS ... 98

12 APPENDIX II - FINAL APPLICATION UML DIAGRAMS ... 99

13 APPENDIX III – CD-ROM CONTENTS ... 101

Chapter

1 Introduction

1

An overview of the project background, aims and structure

1.1 OVERVIEW

As computer systems have become increasingly ubiquitous they have evolved from being isolated entities into large distributed interconnected systems. As a result, software technology has undergone a transition from monolithic systems, designed to run on a single platform, to ad hoc ensembles of semi-autonomous, heterogeneous and independently designed sub-systems. This has lead to the creation of a new classification of software components, entitled *autonomous software agents*. These agents operate on behalf of a user and are often imbued with some form of artificial intelligence allowing autonomous decision making, planning and actions.

In parallel with this paradigm shift in software engineering the amount of electronic data that is generated and the number of services available on-line has increased exponentially. Several next-generation computational services have been proposed that will facilitate the management of and access to the increasing amounts of information and services, such as the Semantic Web (Berners-Lee, Hendler and Lassila, 2001), the Semantic Grid (De Roure, Jennings and Shadbolt, 2001) and the Digital Business Ecosystem project (Di Corinto and Rathbone, 2004). In order to be effective these services will require the use of software agents that are not only capable of automating tasks, but also capable of efficiently coordinating their activities with other agents through cooperation, negotiation and deliberation. However, several problems must be overcome if these visions are to be realised. The agents in the proposed systems will frequently be dealing with incomplete or incoherent information and, in case of multi-agent interaction, conflicts of interest are inevitable. Current agent technology utilises highly formalised classical forms of logic to support their communication and decision-making processes and consequently is unable to effectively deal with these situations.

Argumentation is a relatively new paradigm in Artificial Intelligence, based on a rich flexible framework, which could potentially provide a less brittle form of logic. Recently the use of argumentation in computer science-based applications has focused on formal dialogue systems, allowing structured arguments to be exchanged between agents. Dialogue systems essentially define the principle of coherent dialogue and the conditions under which a statement (commonly referred to as an utterance) made by an agent is appropriate, i.e. the statement furthers the desired outcome or goal of the dialogue. This can be viewed as a game-theoretic approach to dialogues, where speech acts (the statements or utterances) are viewed as moves and a series of semantics present the rules of the game, indicating the conditions that must be satisfied before each utterance is considered to be appropriate. According to the ASPIC Draft Formal Semantics for Communication, Negotiation and Dispute Resolution document (2005) semantics may provide many different functions, but essentially they provide a shared understanding to participants in a communicative interaction of the meaning of individual utterances, of sequences of utterances and of dialogue. Semantics can also provide operational guidelines to designers of agent protocols and to the designers of agents who are using the protocols, presenting a means by which these protocols may be readily implemented and enforced.

1.2 AIMS OF THIS DISSERTATION

The aim of this dissertation is to explore the use of argumentation within agent-based communication and coordination, and more specifically a dialogue game-based scenario where two or more agents (with potentially different interests) are attempting to reach a predefined goal through discourse. Over the past several years an increasing number of theoretical argumentation-based dialogue semantics have been developed and presented for various dialogue types, but their applicability has yet to be demonstrated in a functional agent-based application. Accordingly, this dissertation will investigate and transform the semantics of two dialogue types into a programmatic representation. The dissertation also aims to deliver a set of generic components that is capable of enforcing both the current programmatic representations produced and future representations that support additional dialogue types. A fully functional agent-based dialogue application will also be delivered with the objective being to enforce the dialogue game rules (encapsulated in the theoretical semantics), preventing a participating agent from making an inappropriate move, or utterance, within the game. The first type of dialogue to be investigated, an e-commerce negotiation game, is presented in the ASPIC Draft Formal Semantics for Communication, Negotiation and Dispute Resolution document (2005). The second, a deliberation dialogue game, is

presented by Hitchcock, McBurney and Parsons (2001). It should be noted that the investigations conducted in this dissertation will cover only the rules of the game, i.e. which moves are allowed and will not cover the principles for playing the game well, such as strategies and heuristics.

The final deliverable components produced upon completion of this dissertation will provide:

- Programmatic representations of an e-commerce negotiation and deliberation dialogue semantics, allowing:
    a. The specification of preconditions for each utterance in the dialogue, which indicate when it would be appropriate for an agent to make a particular utterance within an active dialogue.
    b. Additional developers to understand the meaning of each utterance within the dialogue.

- A set of re-usable components to allow generic enforcement of the programmatic representation of the dialogue semantics:
    a. Ensuring that an agent can only make an utterance if the associated conditions in the semantics have been satisfied.

- A distributed application to allow agents to participate in a supervised dialogue game consisting of only valid and appropriate moves (utterances), demonstrating the application of the generic semantic enforcement components. The application will:
    a. Allow two or more agents (human users), who may not be collocated, to participate in a dialogue game.
    b. Manage the state of each agent participating within a dialogue, including the agent's previous utterances and any dialogical commitments incurred as a result of these utterances.
    c. Be constructed of loosely coupled components allowing the agent components, user interface and communication mechanism to be interchanged.
    d. Provide a centralised mechanism to supervise the dialogue, monitoring all utterances made and allowing intervention if necessary.
    e. Utilise the generic semantic enforcement components to ensure that each participant within the dialogue can make only legal moves (i.e. appropriate utterances).

1.3 DISSERTATION STRUCTURE

Chapter 2 aims to familiarise the reader with key concepts and technologies, and illustrate the current state of the art. This chapter can be broadly divided into three themes. Firstly, a historical background and overview of agent-based technologies including coordination and decision-making models is provided. Secondly, contemporary argumentation and formal dialogue systems are introduced. Finally, the purpose of semantics within the context of dialogue systems is discussed and related to the research being conducted in this dissertation.

An overview of current efforts to standardise agent technology and a discussion on how this will impact the design of the demonstration application produced in this dissertation is provided in Chapter 3. Several current agent frameworks are also identified and evaluated, with the ultimate aim of this chapter being to determine the most appropriate technology to implement the demonstration application and semantic enforcement components.

The process of transforming the public axiomatic semantics provided for the e-commerce negotiation dialogue into a programmatic representation is presented in Chapter 4. Chapter 5 continues this work, documenting the design of the first prototype dialogue application that acted as a proof-of-concept for the successful implementation of the semantic representation developed in the previous chapter.

Chapter 6 documents the investigation and creation of the programmatic representation for the deliberation dialogue semantics. This chapter also discusses the changes required to the enforcement mechanism resulting from the investigation of the new semantics and presents the final design of the generic enforcement components.

Chapter 7 presents the implementation of the final demonstration dialogue application, including support for the deliberation dialogue, and also contains details of how the client-side of the application was redesigned into a collection of loosely coupled components, encapsulating functionality and allowing components to be interchanged easily.

The final two chapters of this dissertation critically evaluate the work conducted. Chapter 8 provides a discussion and critique of the results from the research conducted and also evaluates the deliverable components. Chapter 9 provides a conclusion to the research conducted, suggests future work and presents a final evaluation of the work undertaken in this dissertation

Chapter

2 Literature Review

2

Key concepts and a review of the state of the art

2.1 INTRODUCTION

This chapter of the dissertation aims to familiarise the reader with key concepts and technologies, and illustrate the current state of the art. The chapter can be broadly divided into three themes. Firstly, a historical background and overview of agent-based technologies including coordination and decision-making models is provided. Secondly, contemporary argumentation and formal dialogue systems are introduced. Finally, the purpose of semantics within the context of dialogue systems is discussed and related to the research being conducted in this dissertation.

2.2 AGENT TECHNOLOGY OVERVIEW

2.2.1 Trends Leading to the Emergence of Agents

Wooldridge (2002) identifies that the history of computing to date has been marked by five important trends:

- Ubiquity
- Interconnection
- Intelligence
- Delegation
- Human orientation

(Wooldridge, 2002: p.1)

The first trend illustrates that the reduction in cost of computing capability has allowed processing power to become ubiquitous in places and devices that would have been unconceivable twenty years ago. As computer systems have become increasingly ubiquitous they have also evolved from being isolated monolithic entities into large interconnected distributed systems. Genesereth and Ketchpel (1994) and Wooldridge (2002) both discuss that

the trend towards distribution and interconnection has long been recognised as a key challenge within the software engineering domain. Genesereth and Ketchpel (1994) continue by identifying that there is an increasing demand for programs that can interoperate and exchange information and services with other programs, thereby solving problems that cannot be solved alone. However, when interoperability is coupled with the need for a system to represent a user's best interest, fundamental problems can occur. Wooldridge (2002) suggests that it is possible, often highly likely, that the interests represented by one computer system may not be the same as the interests represented in another. It therefore becomes necessary to imbue such systems with some form of artificial intelligence to allow autonomous cooperation, negotiation and resolution of disputes, much as people do in everyday life.

The fourth trend illustrates that the ever increasing intelligence of computer systems has allowed more responsibility to be delegated to them. Modern software applications are now capable of performing relatively complex tasks in a tightly defined domain, such as the autopilot system in modern aircraft, and many people are becoming increasingly happy to give control to these systems. The final trend, human-orientation, suggests that to better model real-world problems there has been a move away from machine-orientated views of programming towards concepts and metaphors that relate to the way humans understand the world.

2.2.2 The Agent Based Paradigm

All the identified trends led to the emergence of a new type of software application classified as autonomous *software agents*. In the classical sense an agent is defined as "One who does the actual work of anything, as distinguished from the instigator or employer; hence, one who acts for another" (Oxford English Dictionary, 2005. Available: http://www.oed.com). A software agent acts on behalf of a user (either a human-user or other software components) and is often imbued with some form of intelligence allowing the agent to work, plan, reason and react to changes in its environment. A precise definition of the term "software agent" has become somewhat of a holy grail within the computer science community and there are several widely accepted definitions. It should therefore be stated that this dissertation will utilise Franklin and Graessar's (1997) definition.

> An autonomous software agent is a system situated within an environment that senses that environment and acts on it, over time, in pursuit of its own agenda and so as to effect[sic] what is senses in the future.
>
> (Franklin and Graessar, 1997: p.4)

An additional interesting property of agents, although often cited as orthogonal (Franklin and A. Graesser, 1996 and Lange and M. Oshima, 1999), is the ability to be mobile network-aware entities which can autonomously change their execution environment, transferring themselves between distributed systems at run-time. As stated by Koukoumpetsos and Antonopoulos (2002), this allows an agent to migrate to remote resources where interaction can take place locally, thereby reducing data traffic and latency, increasing efficiency and making the application more robust. According to Kotz and Gray (1999), code mobility is likely to play an increasingly important role in the future as the bandwidth gap between stationary and mobile devices continues to grow and the amount of data required for processing increases.

In order to realise several proposed next-generation computational services, such as the Semantic Web (Berners-Lee, Hendler and Lassila, 2001) and the Semantic Grid (De Roure, Jennings and Shadbolt, 2001), autonomous software-agents must also be capable of interacting with other agents, and as stated by Wooldridge (2002), "not simply by exchanging data, but by engaging in analogues of the kind of social activity that we all engage in every day of our lives: cooperation, coordination, negotiation, and the like" (p. 1). These multi-agent systems will be composed of multiple individual agents and seem a natural metaphor for building a wide range of what Wooldridge (2002) refers to as "artificial social systems" (p. 1). To facilitate this goal an individual agent must be imbued with some form of standardised coordination model, allowing multiple agents to work together and organise their activities. This requires that each agent contains two fundamental components – a well defined coordination model and an effective decision-making model.

2.2.3 Agent Coordination Models

Gelernter and Carriero (1992) state that coordination models essentially provide operations to create computational activities and to support communication among them. In their seminal paper on coordination languages Gelernter and Carriero (1992) state that "the coordination model is the glue that binds separate [computational] activities into an ensemble." (p. 97). However, coordination is not only about information-exchange, the essence of coordination in the context of the agent-based paradigm is that the information is being exchanged between active agents whose state is constantly evolving and unpredictable. According to Cabri, Leonardi and Zambonelli (2000) this leads to "the choice of coordination model greatly affect[ing] the design of mobile agent applications" (p. 88).

There have been many attempts to classify coordination languages (e.g. Papadopoulos and Arbab, 1998) and recent work by Cabri, Leonardi and Zambonelli (2000) proposed a new taxonomy of coordination models based on the degrees of spatial and temporal coupling effected by a coordination model (defined below).

> - Spatially coupled coordination models require that the interacting entities share a common name space: conversely, spatially uncoupled models enforce anonymous interactions.
> - Temporally couples coordination models imply synchronization of the entities involved: conversely, temporarily uncoupled coordination models achieve asynchronous interactions
>
> (Cabri, Leonardi and Zambonelli, 2000: p. 83)

Figure 2.1 shows the four main coordination model categories proposed from the combination of these characteristics: direct meeting, meeting-orientated, black-board based and Linda-like.

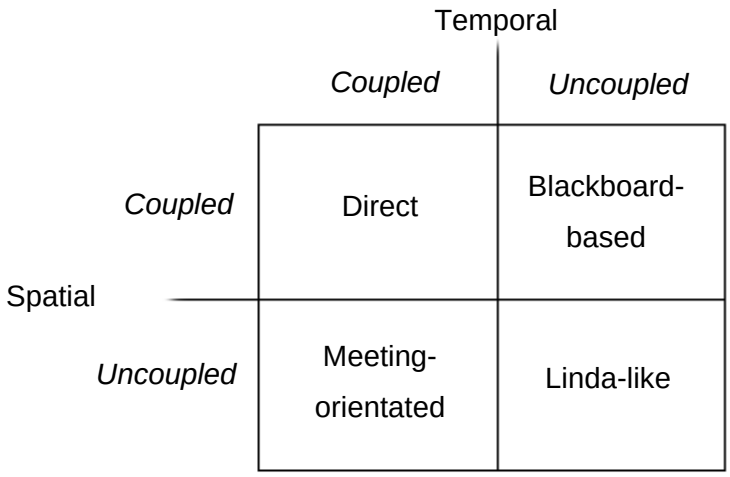


Figure 2.1. Coordination Models for mobile agent applications (adapted from Cabri, Leonardi and Zambonelli, 2000).

The first model, direct coordination, usually implies temporal coupling as the entities involved have to synchronise their communications. Although this form of coordination is often simple to implement (exploiting the client-server method of communication), there are several inherent problems. Firstly, direct coordination models do not always work well in an environment where agents are non-localised, such as Internet applications. This is because repeated interaction requires stable network connections therefore making successful

communication highly dependent on network reliability. Secondly, agents start a communication by explicitly naming the partners involved and therefore remain spatial coupled. The second temporally coupled model, meeting-orientated coordination, partially solves this problem by allowing agents to interact in the context of meetings points, without explicitly naming the partners involved. The main problem with this model is the enforced synchronisation between interacting agents, and due to the unpredictable schedule of autonomous agents, meetings can often be missed.

In the black-board coordination model all interactions are fully temporarily uncoupled as agents interact via shared data spaces (blackboards), using them as common repositories to store and retrieve messages. This means that agents wanting to coordinate activities must agree on a common message identifier to communicate and exchange data (the agent has to read all messages on the entire board and select only the required ones) and therefore they remain spatially coupled to some degree. One advantage offered by black-board based coordination is that hosting environments can easily monitor and control all communications.

The fourth model, Linda-like coordination, uses local tuple spaces as message containers, similar to blackboards. If two agents need to communicate, they generate new data objects (a tuple) and inject this into the tuple space. Tuple spaces are public-read / public-write and any entity in the system can create a new tuple or delete an existing one. In addition, a tuple space bases its access on associative pattern matching mechanisms. This approach enforces full uncoupling, requiring neither temporal nor spatial agreement. Cabri, Leonardi and Zambonelli (2000) argue that associative coordination highly suits mobile-agent applications as agents can utilise pattern-matching mechanisms to deal adaptively with dynamic situations, uncertainty and heterogeneity. The critical problem with Linda-like coordination is that the tuple spaces are publicly accessible, allowing any agent to modify, forge or delete tuples. This can lead to many problems and is inherently insecure in an unrestricted environment such as the Internet. Several refinements to this model have therefore been proposed, such as distributed tuple-spaces, reactive tuple spaces, and law-governed interaction (LGI) which establishes an administrative layer authorising all attempts to read and write according to security and privacy policies.

2.2.4 Agent-based Decision Models

The approach to building artificially intelligent decision-making models for agent-based systems has changed greatly over the past two decades. Although this dissertation will not focus on the exploration or implementation of decision models, this section aims to illustrate how decision making models have influenced the design of communication and coordination mechanisms. Initially deductive reasoning agents where proposed, such as Genesereth and Nilsson's (1987) deliberate agents. In this model an agent can be thought of as a theorem prover where symbolic representations of an agent's environment are logical formulae and the manipulation of these representations corresponded to logical deduction. The next evolutionary stage introduced practical reasoning (Bratman, 1990), allowing reasoning to be directed toward actions. Wooldridge (2002) discusses that practical reasoning is essentially the process of weighing conflicting considerations for and against competing options (depending on what the agent desires and believes) and deciding what action to take. Practical reasoning produced a decision making model that has been used extensively throughout the history of agents, the belief-desire-intention (BDI) framework (Georgeff et al., 1999). This framework relies greatly on planning, and the actions an agent may take are based upon its current state – its beliefs (perhaps atomic facts about the environment), its desires (the design goal or sequence of targets to reach a goal) and its intentions (plans on how the agent intends to achieve its goal).

All of these models have added to the evolutionary process of multi-agent systems and have provided acceptable decision making processes within certain contexts. However, within the context of next-generation agent technology, they are not as robust as they might first appear. As discussed in the ASPIC Theoretical Framework for Argumentation document (2004) "Pertinent information may be insufficient or contrastedly[sic] there may be too much relevant but partially incoherent information. And, in case of multi-agent interaction, conflicts of interest are inevitable" (p. 1). Traditional models for decision making are inherently logic-based, or machine-orientated, and as a result cannot effectively process incomplete or incoherent information. On the other hand, humans excel at processing this type of information by generating structured arguments for and against certain decisions and courses of action and resolving differences through the use of negotiation or deliberation. Accordingly, argumentation may be used to assist autonomous agents by facilitating the exchange and evaluation of interacting arguments which support opinions, claims, proposals and ultimately decisions.

2.3 CONTEMPORARY ARGUMENTATION WITHIN DIALOGUE

2.3.1 The Evolution of Argumentation

Argumentation has been a buzzword in the Artificial Intelligence domain for the last two decades, especially in fields such as nonmonotonic reasoning, inconsistency-tolerant reasoning and natural language processing. However, according to the ASPIC Theoretical Framework for Argumentation document (2004) argumentation has a long history in philosophical literature. Early work began in ancient Greece with Aristotle's structure of argumentation. During the Middle Ages the use of argumentation was also documented among Arabic and scholastic philosophers. In the twentieth century highly influential work was conducted by Toulmin (1958), stating that classical logic, such as mathematical syllogisms (deductive reasoning in which a conclusion is derived from a minor premise and a major premise), were not sufficient to capture the richness of argumentative reasoning, and Hamblin (1970), who used dialogue games to study non-deductive reasoning.

2.3.2 Dialogue Systems and Games

Recently the use of argumentation in computer science-based applications has focused on formal dialogue systems, allowing structured arguments to be exchanged between agents. Dialogue systems essentially define the principle of coherent dialogue and the conditions under which a statement made by an agent (more commonly referred to as an *utterance* in the agent community) is appropriate. A dialogue system therefore specifies when an agent is allowed to make an utterance, with the primary aim being to make an utterance that is appropriate, i.e. furthers the desired outcome or goal of the dialogue. This can be viewed as a game-theoretic approach to dialogues, where speech acts (utterances) are viewed as moves in a game and semantics indicating whether utterances are appropriate at a specified time are formulated as rules of the game.

As the coherence of a dialogue depends on its goal it is important to identify the classifications of various types of dialogue. Walton and Krabbe (1995) have proposed six main categories of human dialogue that are based on the three components; the overall goal of the dialogue, each agents' individual goals and the information each agent has at the start of the dialogue. The categories are:

- *Information-seeking dialogues*, where one participant seeks the answer to some question from another participant.
- *Inquiry dialogues* occur when the participants collaborate to search for a truthful answer to some question.
- *Persuasion dialogues* involve one agent seeking to persuade another to endorse a statement that they currently do not.
- *Negotiation dialogues* consist of the agents bartering over some scarce resource, where potentially each agent's goal is not mutually satisfying with others.
- In *deliberation dialogues* agents collaborate to decide what course of action should be adopted in some situation (this also relates to negotiation dialogues).
- The final classification is *eristic dialogues* where agents argue verbally as a substitute for physical fighting.

This dissertation will primarily focus on exploring the application of two of these categories; negotiation (using an e-commerce based context) and deliberation dialogues.

2.3.3 Elements of a Dialogue System

In order to implement a dialogue system, the core components must be identified and defined. The ASPIC Theoretical Framework for Argumentation document (2004) identifies the common core elements of a dialogue system. Firstly, every dialogue system must have a dialogue goal and at least two participants (agents) who can have various roles. Secondly, dialogue systems also contain two languages, a topic language (representing elements of the dialogue subject) and a communication language (used by agents to correctly communicate utterances within the dialogue). Finally, the central components of a dialogue system are a *protocol*, specifying the allowed moves at each point in the dialogue, *effect rules*, specifying the effects of utterances on the agents' commitments (beliefs), and the *outcome rules*, that define the outcome of a dialogue.

The demonstration dialogue application produced upon completion of this dissertation will utilise all of the previously discussed concepts, with the primary focus being on enforcing the protocol in the generic semantic enforcements components, ensuring that the dialogue consists of only appropriate utterances.

2.4 THE PURPOSE OF SEMANTICS WITHIN DIALOGUE

Agent communication languages, unlike human languages, are formal constructs that are usually defined explicitly. The languages can be thought of as analogous to programming languages, as the agents use them to construct sequences of utterances with which to interact with one another. According to the ASPIC Draft Formal Semantics for Communication, Negotiation and Dispute Resolution document (2005), semantics may provide many different functions, but essentially they provide a shared understanding to participants in a communicative interaction of the meaning of individual utterances, of sequences of utterances and of dialogue. Semantics can also provide guidelines to designers of agent protocols and to the designers of agents who are using the protocols, presenting a means by which these protocols may be readily implemented and enforced.

2.5 SUMMARY

This chapter has provided an overview of the motivation for agent-based technology and identified two important components within agent systems that facilitate interaction, the coordination model and the decision making model. The evolution of decision making models has illustrated several key weaknesses with existing methodologies and highlighted the requirement for more flexible and human-orientated processes, such as argumentation, with which agents can exchange and evaluate information. The chapter has also introduced the core elements of a formal dialogue system and identified that the primary focus of this research will be on enforcing the dialogue protocols with the generic semantic enforcements components. Finally, the concept and purpose of semantics within dialogue systems has been discussed, identifying that they provide a formal framework from which protocols can readily be implemented. Before the application of argumentation-based protocols within dialogue systems can be explored further, appropriate agent technologies and frameworks that will support this goal must be identified, evaluated and compared.

Chapter

3 Comparison of Deployment Platforms

3

A review of existing standards, technologies and frameworks

3.1 INTRODUCTION

This chapter aims to provide an overview of current efforts to standardise agent technology and illustrate how this will impact the design of the demonstration application produced in this dissertation. This chapter will also identify and evaluate several existing agent frameworks with the ultimate aim being to determine the most appropriate technology to implement the demonstration application and semantic enforcement components.

3.2 CURRENT ARCHITECTURES AND STANDARDS

Although agent-based technology has been in development for several years the wide-scale deployment of agent systems has been slow, especially within the commercial environment. Often problems inherent with heterogeneous and distributed technologies are citied as the common factors for the slow adoption, such as issues of interoperability, reliability and security. Many proponents of agent technology, including Bellifemine, Poggi and Rimassa (1999), argue that these problems will continue until well defined standards and architectures are produced and adopted.

> Agent-based technologies cannot realize their full potential, and will not become widespread, until standards to support agent interoperability are available and used by agent developers and adequate environments for the development of agent systems are available.
>
> (Bellifemine, Poggi and Rimassa, 1999: p.1)

Several organisations are working towards the standardisation of agent technologies, the most prominent two being the Foundation for Intelligent Physical Agents (FIPA) and the Object Management Group's Mobile Agent System Interoperability Facility (MASIF). Before evaluating current frameworks to support the research conducted in this dissertation a brief

summary of the two standards will be provided to highlight the current efforts that have been made and identify areas within agent technology that are being addressed.

3.2.1 FIPA

The Foundation for Intelligent Physical Agents (FIPA) (Available: http://www.fipa.org/) is an international non-profit association of companies and research organisations committed to producing specifications of generic agent technologies. The main assumption of the FIPA standard is that only the external behaviour of system components should be specified, leaving implementation details and internal architecture to be specified by the system designers. The current FIPA standard (FIPA97) specifies the roles of several core agents necessary for the management of an *agent platform*; the Agent Management System (AMS) which exercises control over access to and use of the platform, the Agent Communication Channel (ACC) which provides the facilities for communication between agents inside and outside the platform and the Directory Facilitator (DF) agent which provides a look-up directory service for the agent platform.

Allowing a proprietary internal implementation facilitates the potential widespread adoption of this standard, removing restrictions on the technology used and enabling efficient integration with existing systems. However, as demonstrated by Bellifemine, Poggi and Rimassa (1999) the absence of an internal specification creates necessary performance overheads in converting messages being passed to agents external to the system and prevents agents from migrating to a FIPA-compliant platform that is implemented in a different language.

3.2.2 MASIF

The Object Management Group's Mobile Agent System Interoperability Facility (MASIF) (Available: http://www.objs.com/agent/) standard attempts to promote interoperability and system diversity by providing a collection of definitions and interfaces for mobile agent systems (Milojicic *et al*, 1998). Primarily, the MASIF standard defines parameters in an *agent profile* to specify the requirements an agent needs on the receiving agent system. According to Milojicic *et al* (1998) this allows an agent system to support as many agent profiles as its implementation allows. The MASIF standard also allows for multiple programming

languages to be used (to a certain extent), as one of the parameters in the agent profile is *language interoperability.*

MASIF primarily aims to standardise agent management, agent transfer, agent and system names and agent system type and location syntax. The current MASIF standard provides the features required for the first level of interoperability which is the transport of agent information where the information format is standardised. As with the FIPA standard, once the information is transferred from one agent system to another, how the system deals with the parameters internally is an implementation matter and not addressed by the MASIF standard.

3.2.3 Summary of Standards Reviewed

Neither of the architectures reviewed, nor any other current architecture, specifies the true level of standardisation that is needed to become the de facto standard within the agent community. However, this review has identified several interesting points which should be considered when reviewing existing frameworks and designing an agent platform. Primarily, both FIPA and MASIF are concerned with the external interface presented on the agent platform, meaning that internal architectural details are specified as an implementation matter. The deliverables of this dissertation, a programmatic representation of dialogue semantics and a set of enforcement components, are likely to be composed with an agent's internal architecture and therefore not exposed on the external interface. When this fact is combined with the limited timescale available for this dissertation, producing a demonstration agent platform that fully comply with the standards may not be practical, and resources should be diverted to matters more pertinent to the core research. However, it should be stressed that standardisation is not simply being ignored and with careful design of the components their adoption into compliant agents and platforms should be possible at a later date.

3.3 REVIEW OF DEPLOYMENT PLATFORMS

3.3.1 Java – The De facto Agent Technology?

A simple search on the Internet reveals many potential technologies that enable the creation of distributed systems and therefore could be used within this project, for example, Perl, C++, the .NET framework and many other proprietary solutions. However, one particular

technology, the Java J2SE platform, is continually utilised within the agent and mobile code communities. Any review of available resources and potential implementation technologies should always remain unbiased, but it is not difficult to see why this technology is used so extensively. Lange and Oshima (1998) highlight the key properties of the Java platform for the agent-based paradigm as platform-independence, secure execution, multithread programming and object serialization. According to the creator of the Java platform, Sun Microsystems (Available: http://java.sun.com/j2se/index.jsp, 2005), the platform was designed from its origin to operate in heterogeneous networks through the use of a Java Virtual Machine (JVM). This allows the compiler to generate architecture-neutral interpreted byte-code which can be executed on any platform for which a JVM has been created and deployed. The use of Java on internets and intranets also demanded that the platform provide adequate security. Accordingly, the design of the Java security architecture makes it reasonably safe to host any code (for example, an agent) that may not be trusted. The multithreaded programming and object serialization features of the Java language also potentially allow an agent to run in its own lightweight process (allowing autonomy) and be transported over the network when necessary (facilitating code mobility).

Although many frameworks were identified (including implementations not utilising the Java platform), this chapter continues by reviewing four core technologies identified (one of which contains many implementations).

3.3.2 Java Remote Method Invocation (RMI)

The most basic approach to creating a multi-agent system using Java is to utilise the network tools provided in the Java J2SE Development Kit (JDK), such as socket connections or Remote Method Invocation (RMI), without the addition of any existing agent framework. As specified by the Sun Microsystems website, RMI "enables the programmer to create distributed Java technology-based to Java technology-based applications, in which the methods of remote Java objects can be invoked from other Java virtual machines, possibly on different hosts." (Available: http://java.sun.com/products/jdk/rmi/, 2005). Developing agents using this approach would allow complete control over how the system would be designed. However, it would also mean constructing the entire platform and certain concerns such as interoperability and security may not be addressed fully due to the limited timescale of this project.

Utilising the core network tools provided by the Java platform would most likely lead to the creation of a multi-agent system where agents would be both spatially and temporally coupled. This would essentially mean that the coordination model best suited to this technology would be direct coordination. However, with careful planning in combination with complete control over how agents are implemented using this approach, the deliverable enforcement components could be designed to be extensible, allowing future work to provide additional support for different communication mechanisms and coordination models.

3.3.3 Aglets Toolkit

The IBM Aglet agent toolkit supports the creation of lightweight mobile agents that enable the autonomous execution of programs on remote hosts. According to the creators of the Aglets, Lange and Oshima (1998), the Aglets toolkit provides an Applet-like programming model for mobile agents. Applets were incorporated into the original JDK (version 1.0) allowing un-trusted code to be downloaded, on request from a user, from a remote source and executed securely on a local JVM. The Aglet toolkit builds on this framework by allowing Aglets to freely move between multiple Aglet-supporting platforms (not just downloaded once) and to do so autonomously. The framework provided comprises of several key abstractions.

- An Aglet - a mobile Java object that can hop between various aglet-enabled hosts.
- A proxy - This provides an aglet with protection against direct access and can hide the aglet's real location.
- A context - This is an aglets workplace (platform for deployment) and execution environment.
- Synchronous and asynchronous messages passing abilities.
- An identifier for each Aglet that is globally unique.

The toolkit also provides services for maintaining a mobile agent throughout its lifetime such as creation, cloning, dispatching, retraction, activation and deactivation. The majority of services are based on the use of the Java event model, allowing Aglets resident at a context to register event listeners to determine, for example, when they are about to be dispatched or deactivated. The Aglets toolkit essentially uses the direct coordination model as communicating Aglets have to be explicitly named. However, through the use of a context-wide multicast messaging facility, Aglet technology could be utilised to simulate a meeting-

orientated model (where the contexts would become the meeting points) which would spatially uncouple the communicating agents.

The focus of the Aglets toolkit is predominantly based on facilitating mobile agents and accordingly may not be suited to the work proposed in this project which is primarily focused on message-based dialogue. However, the internal architecture of the Aglet platform will provide a useful reference in the implementation phase in this project.

3.3.4 JavaSpaces

The introduction of Linda-like coordination by Carriero and Gelernter (1989) provided a paradigm-shift in distributed computing, allowing a large class of parallel and distributed problems to be implemented easily. Consequently, Sun Microsystems adapted the concept of the tuple-based coordination model into their Jini Technology (an infrastructure for building and deploying distributed systems) and named the implementation JavaSpaces (Available: http://java.sun.com/developer/technicalArticles/tools/JavaSpaces/).

At the core of the JavaSpaces system the "Linda-like" tuples based *associative black board* coordination model is utilised. This decouples the communicating agents both spatially and temporally, essentially allowing messages to be exchanged by injecting them into a shared tuple space for other participants to later read. JavaSpaces also implements an associative lookup mechanism, preventing the need to read the entire contents of the shared tuple-space and allowing searches to be conducted where not all the matching information is known, by utilising wildcards in search criteria tuples named templates. Although this technology provides several very useful features there are several well documented problems. Firstly, JavaSpaces relies on a central server to manage the tuple-space which can become a bottleneck or single point of failure. Secondly, according to Hupter, Freeman and Arnold (1999), there are several caveats in the programming model, such as the inability to declare private instance variables that can be associatively matched, which can lead to problems with both tuple matching and data encapsulation.

3.3.5 Linda-like Variants

In addition to JavaSpaces there exist many other frameworks that add further features to the basic functionality provided by tuple spaces and associative lookup mechanisms. There are

primarily two problems with implementations of the basic Linda-like coordination system. Firstly, the systems often rely on centralised applications located on a single server which can lead to bottlenecks and fault-tolerance problems, and secondly the publicly available tuple-spaces are inherently insecure. Several proposals have been made to remove the reliance on a centralised server, including the distribution of tuple spaces across the network, such as PageSpaces (Ciancarini *et al*, 1997), or the more modern variant Panda (Christian *et al*, 2004) utilising Distributed Hash Tables (DHT), which are essentially self-organising peer-to-peer (p2p) networks, to store tuples. To overcome the problems related to the security of the tuple space there have been several proposed augmentations to the basic model, the first being reactive tuple spaces, such as Tuple Centres Spread over Networks (TuCSoN) (Omicini, 1999) and the Mobile Agent Reactive Spaces (MARS) platform (Cabri, Leonardi, and Zambonelli, 1998). These systems allow the tuple space to be programmed to react to specific actions when agents attempt to access the publicly shared space, for example, allowing inherently unsafe operations to be counteracted. Another solution to the inherent absence of security is Minsky and Ungureanu's (2000) Moses system (Available: http://www.cs.rutgers.edu/moses/), which introduced a mechanism based on law-governed interaction (LGI) for establishing security policies that regulate agent access to tuple spaces. This system makes a strict separation between the formal statement of policy (the law) and the enforcement of the law, which is carried out by a set of distributed policy independent controllers.

3.4 SUMMARY

This chapter began by providing an overview of current efforts to standardise agent technology and illustrated that although the timescale of this project may prevent the demonstration application from being fully compliant with either of the standards, care should be taken when designing the enforcement components to allow composition with compliant implementations in the future. The chapter has also illustrated why Java has become the de facto platform for implementing agent technology and provided an overview of four core implementation methodologies using this platform.

The JavaSpaces implementation of the Linda-like coordination model would provide useful functionality when implementing the demonstration application, as the utterances made within dialogue system are frequently composed as tuples. However, several caveats in the JavaSpaces specification and inherent problems with the model, such as the centralised nature of the service and absence of security, would provide limitations when attempting to

implement the enforcement of semantics. The Linda-like variants, although overcoming some of the previously mentioned issues do not adequately address both problems simultaneously. They also tend to introduce a layer of complexity that will be unnecessary in this research and the learning of proprietary specifications would consume valuable time. The law-governed interaction implementation of Linda, although not directly related to final choice of implementation methodology, has provided many interesting concepts to support the enforcement of communication rules, such as the use of de-centralised controllers to enforce global security policies between communicating agents. The concept that policies should be separate from the enforcement mechanism also supports the initial specification of the deliverable generic components in the introduction of this dissertation.

The Aglets framework is adept at promoting code mobility, but its message passing abilities are encapsulated within a proprietary specification and are therefore somewhat limited. Accordingly, this will not facilitate the easy integration of the enforcement mechanisms. In addition, due to previous experience of working with the Aglets toolkit, debugging agent program code can often be inherently difficult, due to almost all Aglet application programmers interface (API) methods being designed to throw non-specific Java Exceptions (such as the infamous NullPointerException) which provide minimal debugging information. Often the most effective way to track down problems within an Aglet agent is to insert statements between every line of code to print to the command line the current internal status, which is laborious at best.

Although RMI is essentially the most simplified version of all the technologies reviewed here it will allow all of the agent components and the hosting platform to be built to a custom specification. Care must be taken to design the system to be as extensible as possible and concepts taken from the design of the Aglet toolkit and the implementation of law-governed interaction will facilitate this aim. The demonstration application produced will most likely have to utilise direct coordination with agents being explicitly named and all communication synchronised. With the underlying implementation technology determined the next stage in the development of the demonstration dialogue application will concentrate on the design of the programmatic representation of the dialogue semantics.

Chapter

4 Creating the Programmatic Representation

4

Transforming the theoretical semantics

4.1 INTRODUCTION

This chapter documents the transformation of the theoretical e-commerce negotiation dialogue semantics into a programmatic representation to allow enforcement of the protocols in a dialogue game. The programmatic representation must provide two core functions. Firstly, it should facilitate enforcement of the conditions which indicate when it would be appropriate for an agent to make each utterance within the dialogue. Secondly, the representation should also facilitate future developers to understand the meaning and preconditions of each utterance within the dialogue. To reduce the initial complexity of the investigation, only the first series of semantics will be transformed in this chapter, leading to the creation of a proof-of-concept prototype dialogue application in the next chapter. If the implementation of the semantic representation proves successful, the additional deliberation dialogue representation and generic enforcement components will be created accordingly.

4.2 E-COMMERCE DIALOGUE OVERVIEW

The e-commerce semantics were presented as part of a prototypical dialogue-based scenario within the ASPIC Draft Formal Semantics for Communication, Negotiation and Dispute Resolution document (2005). The dialogue involves an on-line discourse between two agents, one acting as a potential consumer of some product and one representing a potential vendor of the product. Their discussion concerns negotiating the trade-offs made between different product features and both participants may be making decisions. The potential consumer is accepting or rejecting feature bundles from the vendor, while the vendor is accepting or rejecting feature bundles proposed by the consumer. To achieve resolution of the e-commerce negotiation dialogue, the buyer must agree to purchase a particular product and the seller must also agree to sell the product. Either of the participants may also terminate the dialogue at any time.

4.3 SYNTAX AND SEMANTICS OF THE E-COMMERCE DIALOGUE

The syntax of the e-commerce dialogue allows the participation of two agents within the dialogue, which can be human or autonomous entities. However, in order to reduce to the complexity of the prototype application, it is assumed that both agents in the dialogue will be operated by human users, eliminating the need to implement a complex decision-making process. An overview of the syntax for the e-commerce negotiation protocol is presented in Figure 4.1, with a comprehensive discussion available in the ASPIC Draft Formal Semantics for Communication, Negotiation and Dispute Resolution document (2005). This document also contains a public axiomatic semantics for the e-commerce negotiation protocol which presents preconditions and post-conditions for the six types of locutions defined in the syntax, an example of which can be seen in Figure 4.2. The subject-matter of dialogue can be represented in a prepositional language by lower case Roman letters and participating agents are denoted by P1, P2, etc.

Participants: There are two participants, a potential Buyer and potential Seller.

Dialogue Goal: The Buyer may seek information about product offerings, persuade the seller to offer a particular product or purchase a product provided by the seller.
The seller may seek to provide information about products, learn about the buyer's requirements or persuade the buyer to purchase a particular product.

Communication Language: The minimum locutions needed for a dialogue between buyer and seller of the type described in this scenario are:
OPEN-DIALOGUE: Both the Buyer and Seller must be able to indicate a willingness to enter into a dialogue at this time on this topics with the other party
PROPOSE-TO-SELL(θ): Seller must be able to propose a product offer, θ, to the buyer for purchase at this time
REQUEST-TO-BUY(θ): Buyer must be able to request a product offer, θ, for purchase from the Seller at this time.
ACCEPT-TO-BUY(θ): Buyer must be able to indicate a willingness to purchase a proposed product offer, θ, from the Seller at this time.
ACCEPT-TO-SELL(θ): Seller must be able to indicate willingness to sell a proposed product offer, θ, to the Buyer at this time.
END-DIALOGUE: Because they are autonomous entities, both Buyer and Seller may leave the dialogue at any time.

Figure 4.1. Overview of syntax for the e-commerce negotiation protocol.

REQUEST-TO-BUY

Syntax: REQUEST-TO-BUY(P_j, θ)

Meaning: Agent P_j indicates a willingness to purchase the product bundle represented by θ at this time.

Pre-conditions:

1. Agent P_j has previously uttered OPEN-DIALOGUE(P_j, P_i, *L*) in this dialogue.
2. No other agent P_i has previously uttered PROPOSE-TO-SELL(P_i, θ) in this dialogue
3. Neither agent P_i nor agent P_j has uttered END-DIALOGUE(_)

Figure 4.2. Example of the e-commerce dialogue semantics.

As Figures 4.1 and 4.2 show, the locutions are represented as tuples, predicated by the utterance type. This allows utterances to be communicated using a variety of coordination models, and would specifically suit the associative blackboard coordination model. The tuple contents of the majority of locutions include the identity of the agent from which the tuple originated and the identity of the agent for which it is intended. Although, the syntax expresses that the dialogue may only contain two participants, this information would allow tuples to be identified by the appropriate participants if the tuple space in which the utterances were to be injected was shared with more agents. However, if the two agents are using the direct coordination model, as determined by the use of RMI in the prototype application, this information could be considered redundant.

4.4 REPRESENTING THE SEMANTICS

4.4.1 Identifying the Required Information for Semantic Enforcement

The first task when designing the programmatic representation of the semantics was to identify the information that will be required to determine whether the preconditions of a locution have been satisfied. The REQUEST-TO-BUY locution (the semantics of which are shown in Figure 4.2) illustrates a typical example of the type of conditions that must be satisfied. The majority of preconditions in the semantics specify that either the agent who wants to make the utterance or another agent participating in the dialogue must (or must not) have previously uttered a specific utterance type. For example, for an agent to make an appropriate REQUEST-TO-BUY utterance, the agent must have previously uttered OPEN-DIALOGUE and no other agent may have uttered either PROPOSE-TO-SELL or END-DIALOGUE. Accordingly, this identified that each agent participating in the dialogue must store all of the previous utterances to represent the current state of the dialogue. As Figure

4.2 shows, the tuple content of the new utterance is also used to match the content of previous utterances. For example, the preconditions of the REQUEST-TO-BUY(Pi,θ) utterance specifies that any agent, represented by Pi, must not have uttered the locution PROPOSE-TO-SELL(Pi,θ) with θ representing a specific product or product bundle. This required that the tuple content of the new utterance should be extracted and compared with previous utterances to see if a match is found (i.e. both agents are referring to the same product or product bundle). This would be easy to implement using an associative lookup mechanism, but as the prototype is being designed using no existing framework that supports this function, implementation would be difficult (the implementation of systems that support this mechanism such as JavaSpaces is highly complex). Therefore, at this stage of the research it was decided that only the previous utterance types would be compared and not the tuple content in order to determine if the preconditions had been satisfied. This should not have a big impact on the semantics within the e-commerce dialogue as the majority of information specified in the tuples is the identity of the uttering agent and the intended recipient which, as stated previously, is implicit in a dialogue consisting of two participants utilising direct coordination.

4.4.2 Designing the Utterance Class

With the required information identified the next step was to design a class that would provide a programmatic representation of an utterance. The simple Utterance class (shown in Figure 4.3) was created.

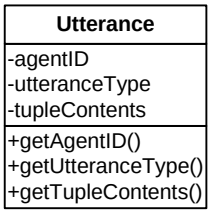


Figure 4.3. Utterance UML class diagram.

To facilitate the comparison of previous utterance types a mechanism was needed that would be more type-safe than the comparison of string representations (for example, it would be easy to miss-type REQUEST-TO-BUY in the application code, which would prevent utterances being correctly matched. Consequently, this would lead to unexpected behaviour that was hard to debug). Accordingly, Java's equivalent of global variables, *public static final*

variables, was utilised to provide an integer representation of each utterance type allowing type-checking to be enforced more effectively. The enumeration feature added to Java 5.0 (the Enum class) was also considered to solve this problem, but the integer solution was easier to implement, functioned correctly on previous versions of Java and future work in the dissertation highlighted that using Enum would have restricted the ability to make polymorphic calls on a method where utterance types were being passed as parameters (which will be discussed further in Chapter 6).

4.5 THE CONCEPT OF AN ENFORCEMENT MECHANISM

One of the main aims of this dissertation is to provide a generic set of re-useable components to allow enforcement of the semantics. Ultimately the components must expose a well-designed public interface allowing the programmatic logic to be encapsulated and hidden from the other implementing components. The dissertation deliverables also specify that the enforcement components themselves must be generic, and therefore loosely coupled, to enable the mechanism to enforce many different dialogue semantics (i.e. allowing programmatic representations of multiple semantics to be interchanged with a standard enforcement mechanism). The law-governed interaction (LGI) mechanism that was identified in the literature review attempts to solve a similar problem. Minsky and Ungureanu (2000) argue that in distributed systems coordination policies need to be enforced and that these policies need to be separate from the enforcement mechanism, allowing multiple policies to be both implemented simultaneously and deployed incrementally. The policies in LGI, although intended to enforce security, essentially specify a series of rules for coordination between agents and therefore can be seen an analogous to the dialogue semantics, which provide rules for dialogical interaction. In LGI all outbound communications from an agent are subject to the law, which is in essence simply a function that returns true or false depending on whether the communication is valid and allowed at this time. Accordingly, this can easily be adapted to the context of an agent making an utterance, where a mechanism could return true or false depending on whether the associated preconditions have been satisfied, indicating the utterance is appropriate. Section 4.6 in this chapter discusses how such a mechanism was developed

4.5.1 Designing the Semantic Policy

Freeman and Freeman (2004) discuss that in good object-orientated design components should be programmed to interfaces, not implementations, and therefore it makes sense to first specify the semantic representation as an interface which can later be implemented. Accordingly, the next stage in the design of a programmatic representation was to transform the public axiomatic semantics provided into an interface-based contract. The Java platform provides *interfaces* for exactly this purpose, allowing pure virtual methods to be specified and enforced in any implementing class. In addition there are several design methodologies that would support this implementation, for example, the Design by Contract (DBC) software engineering methodology, pioneered by Meyer (2000), is a widely acknowledged contract-based technique for writing reliable software. Two of the key elements of DBC are preconditions and post-conditions, exactly as presented in the public axiomatic semantics provided. However, the Java programming language does not directly support DBC. Extensions to the language that provide this functionality are available, such as *jmsassert* (Available: http://www.mmsindia.com/DBCForJava.html, 2004), but they tend to increase the complexity of the code unnecessarily, are often platform independent or rely on the catching of AssertionExceptions, which according to Sierra and Bates (2002) is generally not recommended.

4.6 ADAPTING THE DESIGN BY CONTRACT METHODOLOGY

4.6.1 The Assert Mechanism

Although the DBC concept has not been implemented to a satisfactory standard on the Java platform, the design of this methodology provided an interesting concept for exploration. The assertion mechanism present in the Java language since version 1.4, allows the programmer a "limited form of design-by-contract style programming during the development phase" (Available: http://java.sun.com/developer/technicalArticles/JavaLP/assertions/, 2005). This is achieved by throwing exceptions and halting the program execution if a specified condition in a method is not satisfied. For example, if the preconditions of a hypothetical method named loadFile guaranteed that the filename was not null, the assertion mechanism could be used to enforce this behaviour, as demonstrated in Figure 4.4. In this example, if the filename parameter string was null the condition in the assertion statement would evaluate to true and an AssertionException would be thrown, halting program execution.

```
/**
* loadFile – loads a file into the application
* @param fileName the name of the file
* Precondition: the fileName string must not be null
*/
public void loadFile(String fileName) {
        assert filename != null : "File cannot be null";

        //the complete code logic continues here…
}
```

Figure 4.4. Using assert to enforce preconditions.

If another hypothetical function, named previousUtteranceMade, could be created that could determine whether a particular type of utterance had previously been made, then the example above could easily be adapted to determine whether an utterance is appropriate. For example, the preconditions in the e-commerce REQUEST-TO-BUY locution state that the agent attempting to make this utterance must have previously uttered the OPEN-DIALOGUE locution and that no other agents have uttered PROPOSE-TO-SELL or END-DIALOGUE The pseudocode in figure 4.5 illustrates how the assertion mechanism could be utilised.

```
/**
* Preconditions Agent Pj has previously uttered OPEN-DIALOGUE in this
* dialogue.
* No other Agent Pi has previously uttered PROPOSE-TO-SELL in
* this dialogue.
* No other Agent Pi has uttered END-DIALOGUE.
*/
public void utteranceAppropriateRequestToBuy() {
assert previousUtteranceMade(thisAgent, OpenDialogue) == true :"OPEN-DIALOGUE not
uttered";
assert previousUtteranceMade(otherAgent, RequestToBuy) == false : "REQUEST-TO-BUY has
previously been uttered";
assert previousUtteranceMade(otherAgent, EndDialogue) == false : "END-DIALOGUE has
previously been uttered";


//at this point the utterance is appropriate
//and the code logic continues here…
}
```

Figure 4.5. Using assert to enforce dialogue semantic preconditions.

Figure 4.5 seems to demonstrate the solution to semantic enforcement. However, assertions are designed to be utilised only during development time and should be disabled when the application is deployed. The Sun Developer website (2005) states that "[assertion] messages are for us -- developers -- and not users" (Available: http://java.sun.com/developer/technicalArticles/JavaLP/assertions, 2005). Assertions were also designed to completely halt the execution of program if a condition has not been satisfied and therefore they are of limited use in the context of dialogue which could contain many utterances.

4.6.2 Adapting the Assert Mechanism

Although the assert mechanism could not be used directly, the concept was adapted and instead of throwing a program-halting exception a Boolean test was utilised, with the result of the evaluation indicating whether the utterance was appropriate. Figure 4.6. illustrates how this method essentially becomes a function and has obvious similarity with the LGI law enforcement discussed previously.

```
/**
* Preconditions Agent Pj has previously uttered OPEN-DIALOGUE in this
* dialogue.
* No other Agent Pi has previously uttered PROPOSE-TO-SELL in
* this dialogue.
* No other Agent Pi has uttered END-DIALOGUE.
*/
public boolean utteranceAppropriateRequestToBuy() {

boolean utteranceAppropriate = false; //default not appropriate
if (previousUtteranceMade(thisAgent, OpenDialogue)  == true &&
        previousUtteranceMade(otherAgent, RequestToBuy) == false &&
        previousUtteranceMade(otherAgent, EndDialogue) == false) {

        utteranceAppropriate = true;
}
return utteranceAppropriate;
}
```

Figure 4.6. Using Boolean values to enforce dialogue semantic preconditions.

At this stage of the design almost all the information required for the construction of the contractual interface had been identified. The implementation details of the previousUtteranceMade method that determines whether an utterance type has previously been uttered will be specified in the prototype concrete implementation of the semantic interface. However, from the example code it is evident that both the previous utterances of the current agent and the other agent participating in the dialogue will be required in the method body and therefore these must be specified as parameters in the interface method signature. Accordingly, appropriate data structures will need to be specified to store the previous utterances.

4.6.3 Storing the Previous Utterances

As there can only be two agents in the e-commerce dialogue the most efficient way to store the previous utterances was to utilise two ArrayLists (essentially a dynamic array with support for fast random access). As utterances are made or received by the agent they can be stored in the agent's own previous utterance ArrayList or the additional agent's ArrayList respectively, allowing both participants in the dialogue to maintain a complete history of

utterances. The implementation details of how the semantic enforcement mechanism will check if an agent has previously made an utterance is discussed in the next chapter.

4.6.4 Documenting the Preconditions and Post-conditions

The final challenge when developing the semantic interface was informing the developer of the meaning of each utterance and the associated preconditions and post-conditions. The method ultimately chosen was provided by the Javadoc tool (Available: http://java.sun.com/j2se/javadoc/, 2005), which is included with the Java platform. Javadoc allows a programmer to specify API documentation, such as the parameters or guaranteed functionality of a method. Although preconditions and post-conditions are not yet fully supported in Javadoc, they can be included in the source code and in combination with the implicit contract specified by the interface, will provide a reasonable guide to the semantics for a developer that wants to implement the e-commerce dialogue negotiation protocol.

4.7 THE SEMANTIC INTERFACE

A Java interface named EcommerceDialogueSemantics was created and the method signatures (taking two ArrayList's as parameters and returning a Boolean value) and Javadoc was specified accordingly. As stated previously, the contractual interface only needs to define the method signatures and the concrete implementation can be delegated to the implementing semantic enforcement component. An appropriately named virtual method was created for each locution specified in the syntax with the requirement that the concrete implementation methods will return true if the utterance is appropriate at this time (i.e. all of the specified pre-conditions have been satisfied) or false it is not. Figure 4.7 shows an example of the REQUEST-TO-BUY semantics transformed into the contractual interface programmatic representation that will be utilised within the prototype distributed dialogue application.

```
/**
* REQUEST-TO-BUY utterance - Syntax REQUEST-TO-BUY(Pj, 0)
* <p>Meaning: Agent Pi indicates a willingness to purchase the
* product bundle represented by 0 at this time</p>
*
* @pre Agent Pj has previously uttered OPEN-DIALOGUE in this
* dialogue.
* No other Agent Pi has previously uttered PROPOSE-TO-SELL in
* this dialogue.
* Neither agent Pi or Pj has uttered END-DIALOGUE.
*
* @post No post-conditions
*
* @param agentUtterances the previous Utterances uttered in this
* dialogue by the agent now uttering
* @param otherAgentUtterances the previous Utterances uttered by
* other agents participating in this dialogue
*
* @return true if the utterance is appropriate at this time
*/
public boolean requestToBuy(ArrayList <Utterance> agentUtterances,
                            ArrayList <Utterance> otherAgentUtterances);
```

Figure 4.7. Semantics encapsulated within the DialogueUtterances interface and Javadoc.

4.8 SUMMARY

This chapter began with an overview of both the syntax and semantics of the e-commerce dialogue and continued by illustrating each stage in the transformation of the original semantics into a programmatic representation. The chapter concluded with the creation of the programmatic representation encapsulated within the EcommerceDialogueSemantics interface. The next stage in the prototype application development will be to design a semantic enforcement mechanism that will provide a concrete implementation of the virtual methods specified.

Chapter

5 The E-commerce Dialogue Prototype

5

Proof-of-concept for the semantic representation

5.1 INTRODUCTION

This chapter details the design of the first prototype dialogue application that acted as a proof-of-concept for the successful implementation of the semantic contract developed in the previous chapter. The chapter begins with an overview of the application components developed and then continues to discuss the design decisions made with each component.

5.2 PROTOTYPE OVERVIEW

The prototype application presented in this chapter consists of six core components:

- The programmatic representation of the semantics, presenting preconditions and post-conditions, encapsulated in a Java Interface *EcommerceDialogueSemantics*. (Documented in the previous chapter)
- A Web-based Graphical User Interface (GUI) allowing remote users to participate in a dialogue and enforcing the semantic interface, provided by the *ArgueApplet* Java Applet.
- A programmatic representation of utterances made by each agent that can be transported across the network, provided by the serializable *Utterance* class.
- A server that manages client registration and allows agents to make and receive utterances within the dialogue. This is provided by the *RMIGuardianAgent* class which permits connection from remote clients using the Java Remote Method Invocation (RMI) API.
- Logging and server-side display of all utterances made within the dialogue, provided by the *UtteranceLogBean* JavaBean.
- Persistent storage in a relational database of the utterances made within every dialogue, provided by the *DataBaseBean* JavaBean.

These components were grouped into three Java packages; *client*, containing all client-side classes, *server*, incorporating the server-side components and *common*, including classes common to both client and server (Appendix I contains UML class diagrams of all three

packages.) The remainder of this chapter summarises the design and implementation of each of the six components.

5.3 ENFORCING THE SEMANTICS

With the creation of the contractual semantic interface documented in the previous chapter, the first decision in the design of the prototype application was where to place the concrete implementation of the semantic enforcement mechanism components. The main criteria for the successful implementation of the semantic enforcement components were (1) they must be able to intercept utterances before they are incorporated into the current dialogue and (2) they must have access to the previous utterances of both the agent attempting to make the utterance and the additional agents participating in the dialogue. As the prototype application is utilising RMI for communication and a central server to monitor the dialogue, the enforcement mechanism could be deployed in either of two locations; the server-side or the client-side.

5.3.1 Server-side Deployment

There are several strong arguments for the server-side deployment of a centralised coordination mechanism. For example, only one copy of the previous utterances would need to be stored, the semantic interface would only need to be deployed and managed at one location and security would be guaranteed (provided the server was considered trusted).

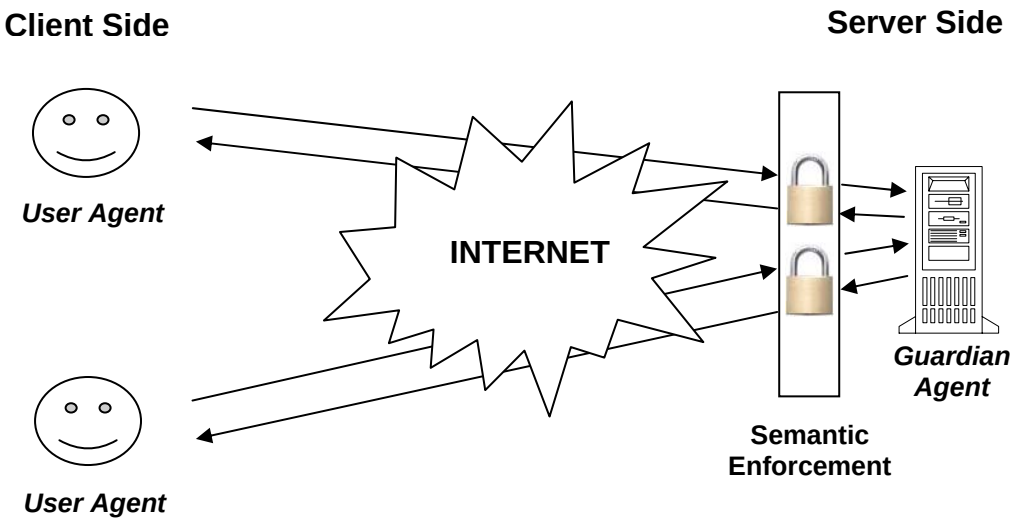


Figure 5.1. Server side semantic enforcement.

However, the major disadvantage of the centralised coordination mechanism is that it is not scalable. Minsky and Ungureanu (2000) argue that a centralised coordinator can become a bottleneck and a dangerous single point of failure. Minsky and Ungureanu (2000) continue by arguing that an individual agent should be responsible for enforcing the policy locally because "a single software agent operating within such a system may find itself interacting with several groups of agents operating under disparate policies" (p. 3). This is definitely true in a dialogue system where potentially an agent could be participating in a variety of dialogue types. Minsky and Ungureanu (2000) conclude by stating that "the enforcement needs to be decentralised" and in this application this would mean distributing the enforcement mechanism to the location of each agent, i.e. the client-side.

5.3.2 Client-side Deployment

As stated previously, Minsky and Ungureanus' (2000) LGI model advocated installing distributed controllers between the client and the communication medium, therefore reducing the computational burden on the server and unnecessary use of the communication medium. As this model recommends a distributed mechanism the system scales well as there is no reliance on a single enforcement component.

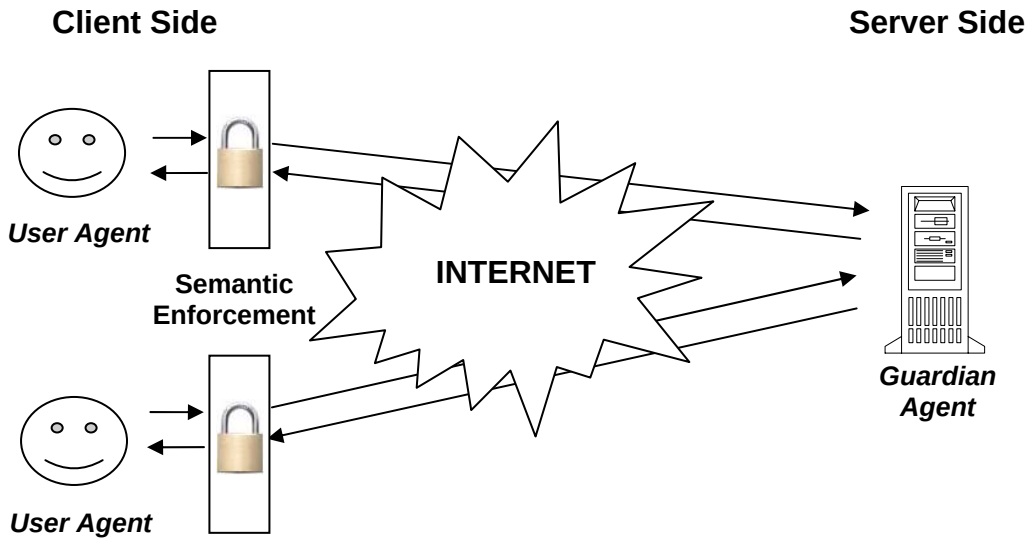


Figure 5.2. Client side semantic enforcement.

There are several caveats to pushing the responsibility of enforcement onto the client, primarily can a client be trusted not to alter, intentionally or otherwise, a controller?

Mechanisms to ensure the integrity and validity of a controller have been recommended by Minsky and Ungureanu (2000), and range from impractical for the average system, such as running all controllers on a secured and trusted processor (resistant to tampering), to the more feasible method of utilising Public Key Cryptography and digital signatures. A discussion of the various merits and implementation details of each mechanism is outside the scope of this dissertation, but it is an important point to be considered if such a system were to be deployed in an environment were the integrity of agents could not be guaranteed.

5.3.3 Choosing the Deployment Location

Research conducted clearly indicated that to increase efficiency and scalability the semantic enforcement mechanism should be pushed onto the client side components. Although the final set of components produced will provide a generic enforcement mechanism (which will be loosely coupled with the client interface), in order to reduce the complexity of the initial prototype the decision was made to incorporate the enforcement mechanism directly into the client-side user interface component. The following section of this chapter details how this was achieved.

5.4 THE ARGUEAPPLET

5.4.1 Component Overview

The client-side graphical user interface (Figure 5.3) is provided by the ArgueApplet class which extends the JApplet class, allowing the code to be downloaded and executed in any recent Java-compliant Web browser. The ArgueApplet class also implements the EcommerceDialogueSemantics interface specified in the previous chapter and provides a simple enforcement mechanism that ensures that only appropriate utterances are sent to the server to be included in the current dialogue. Each human participant in the dialogue must operate a unique instance of the ArgueApplet class as this acts as an agent for the user and is responsible for maintaining the current state and also transmitting and receiving any utterances made in the dialogue.

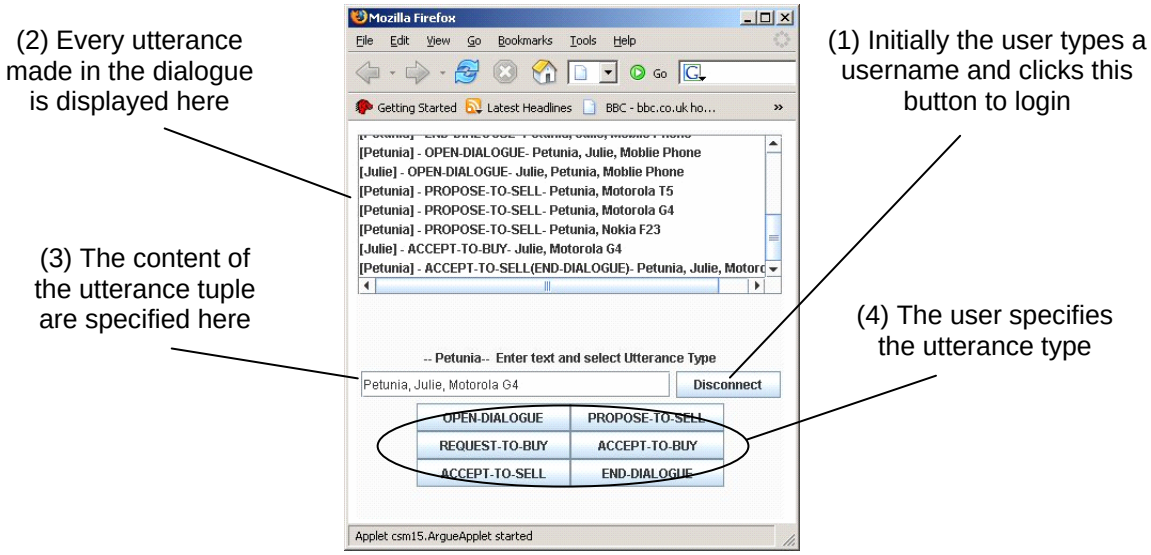


Figure 5.3. GUI constructed using Swing and the GridBagLayout layout manager.

5.4.2 Making an Utterance

In order to participate in a dialogue a user must first enter a user name in the appropriate text box and clicks the 'Connect' button (see 1 in Figure 5.3). If the server determines that the user name is valid and unique the dialogue window (2) at the top of the display informs the user the login was successful and they may now participate in a dialogue. At this stage the user has two choices, they may either create a new dialogue session (by uttering OPEN-DIALOGUE) or join an existing dialogue (in the e-commerce dialogue, this is also indicated by uttering OPEN-DIALOGUE).

As Figure 5.3 shows, when the user is participating in a dialogue they are presented with a dialogue window displaying all utterances made within the currently active dialogue (2). The user may attempt to make an utterance at any time by typing the contents of the tuple into the text box (3) and then clicking on the appropriate button (4) located at the bottom of the interface, indicating the utterance type. The ArgueApplet class then encapsulates the utterance into a serializable *Utterance* object (discussed in the next section) and passes this to the semantic enforcement mechanism. It should be noted that the prototype performed minimal syntax checking, as the semantic enforcement system only has to check the type of utterances being made (and not the tuple contents) to determine the preconditions that must be satisfied.

5.4.3 Integrating the Utterance Class

The Utterance class specified in the previous chapter required minor modifications as the prototype application highlighted that this class will be used in many different roles. Primarily the class will be validated by the semantic enforcement mechanism and then transported over the network to the server. Accordingly, the class had to implement the Serializable marker interface, indicating that an instance of this object could be converted to an Object Stream before being sent over a network connection. Initial class fields were developed utilising primitive variable types and the String class (all of which are implicitly Serializable). These included the client ID (an integer representing the agent making the utterance), the utterance type (an integer representing the interface utterance type) and the contents of the utterance tuple (a String representation of the tuple e.g. “Daniel, John, Mobile Phones”).

5.4.4 Enforcing the Semantic Policy

In addition to providing the user interface, the ArgueApplet also acted as the semantic enforcement mechanism and therefore needed to maintain a store of previously made utterances for both the current user and any other participants in the dialogue. As stated in the previous chapter, the most efficient way to store utterances for the two participants engaged in dialogue was to create two type-safe ArrayLists (type-safety ensures only Utterances can be stored in the collection and is denoted by ArrayList<Utterance> in the sample code shown). When an utterance is made by either the current agent or received from the server, it was placed into the appropriate ArrayList, maintaining a complete history of all utterances in the dialogue.

During the design of the semantic interface (section 4.5.2), it was stated that the concrete implementation would need to utilise a method that was capable of checking an ArrayList for a previous utterance type. Accordingly, a utility method named containsUtteranceType was created to replace the pseudocode previousUtteranceMade method. The containsUtteranceType method takes two parameters, the ArrayList of utterances to be checked (either the agents’ own or the other participants’) and an integer representation of the utterance type for the current dialogue. Figure 5.4. illustrates the code contained within this core method.

```
    /**
     * Determine if utterance is stored in the specified commitment store
     *
     * @param messageStore The commitment store
     * @param messageType The specified message type
     * @return true If the utterance is found
     */
    private boolean containsUtteranceType(ArrayList<Utterance> utterances,
                                          int utteranceType) {
        if(!utterances.isEmpty()){
            for (Message tempUtterance : utterances) {
                if (tempUtterance.getUtteranceType() == utteranceType) {
                    return true;
                }
            }
        }
        return false;
    }
```

Figure 5.4. Code excerpt showing the core method containsUtterance.

With this core method defined, the contractual semantic interface was easily implemented in the ArgueApplet class. Figure 5.5. illustrates an example of how the adapted concepts taken from the DBC methodology in combination with the containsUtteranceType method allowed a concrete enforcement method to be created. This method checks if the REQUEST-TO-BUY utterance is appropriate using the two lists of previous utterances passed as parameters to indicate the current state of the dialogue.

```
/**
* Concreate implementation of DialogueUtterance utterances
* @param agentUtterances the previous Utterances uttered in this
* dialogue by the agent now uttering
* @param otherAgentUtterances the previous Utterances uttered by
* other agents participating in this dialogue
*
* @return true if the utterance is appropriate at this time
*/
public boolean requestToBuy(ArrayList<Utterance> agentUtterances,
                            ArrayList <Utterance> otherAgentUtterances){
   if ((containsUtteranceType(agentUtterances, DialogueUtterance.OPEN_DIALOGUE) &&
containsUtteranceType(otherAgentUtterances, DialogueUtterance.OPEN_DIALOGUE))
            && (!containsUtteranceType(otherAgentUtterances,
DialogueUtterance.PROPOSE_TO_SELL) &&  !containsUtteranceType(agentUtterances,
DialogueUtterance.PROPOSE_TO_SELL))) {
      return true;
   }
    return false;
}
```

Figure 5.5. An example of a concrete implementation of the semantic interface utterance appropriate methods.

5.4.5 An Example of the Enforcement Process

As stated previously, when a user wants to make an utterance he or she types the tuple contents into the text box and clicks the appropriate utterance type button. When an utterance type button is clicked an ActionEvent is fired and the inner class HandleMessageTypeButtons ActionPerformed method is called, which handles the event by initiating a sequence of method calls. Firstly, the utterance type is identified using a switch statement (Figure 5.6). Secondly, the appropriate method from the concrete implementation of the semantic interface is called (shown in bold in Figure 5.6) and the result stored in the utteranceAppropriate Boolean variable. Finally, if the method returns true, this indicates the utterance is appropriate at this time and the ActionPerformed method sends this utterance to the server for inclusion in the dialogue.

```
public void actionPerformed(ActionEvent e) {
  boolean utteranceAppropriate = false;

    //check if the utterance is appropriate
    switch(buttonID) {

    //check for additional utterances omitted in this Figure

    case EcommerceDialogue.REQUEST_TO_BUY:
       utteranceAppropriate = requestToBuy(thisPreviousMessages,
                                                    otherAgentPreviousMessages);
       break;
    }

    if (utteranceAppropriate) {
        //send utterance to server
       //Communication code omitted in Figure
       }
}
```

Figure 5.6. Using the implementation of the semantic interface to determine if the user selected utterance is appropriate.

5.4.6 Implementing the Communication Mechanism

The server component (documented fully in the next section) was designed to utilise a client-push/client-pull methodology for the sending and receiving of Utterances. When an utterance has been determined appropriate it is sent to the server component for inclusion into the current dialogue and to be broadcast to additional participating agents. This is achieved by invoking the send method on the server, using RMI, with the new utterance included as a parameter. As the utterance class implements the Serializable interface, the utterance is simply converted to a byte stream and sent across the network to the server component. To receive new utterances the ArgueApplet class must continually poll the server checking for

new utterances. To allow the GUI to remain responsive to user input, an additional thread is spawned in the ArgueApplet class which continuously polls the server every 1 second.

5.5 THE RMIGUARDIANAGENT SERVER

5.5.1 Component Overview

The server-side component, named RMIGuardianAgent, acts as a server and "Guardian Agent" to all participants within the dialogue. The server manages all the clients, allowing users to log in and participate in a dialogue. A client receives an AgentUser object from the server upon successful login which is essentially an identity token, containing a client ID variable allowing an agent to prove they have been authenticated. As the AgentUser and client ID are passed over the network in plaintext this is inherently not secure, but could be modified in the future, either by securing the communication mechanism or by utilising cryptographic techniques, such as digital signatures. Clients are prevented from interacting directly and all communication is routed through this server, allowing utterances to be monitored and recorded in persistence storage. The RMIGuardianAgent could also potentially act as a mediator or an authority for the dialogue, but this is not currently implemented.

5.5.2 The Guardian Agent Implementation

The implementation of the RMIGuardianAgent is divided into two key components. The first component, named RMIGuardianAgent, is an interface that exposes methods that may be called by a remote client. Due to the potential for communication failure in the network, the RMI specification insists that every method declared within this interface must throw a RemoteException which, in this application, is handled by the ArgueApplet class. The second component, named RMIGuardianAgentImpl, is a concrete implementation of the interface that is responsible for registering the class with the RMI registry and providing the functionality of the methods that may be invoked remotely. Figure 5.7. illustrates how remote clients establish a connection and communicate with the RMIGuardianAgent server.

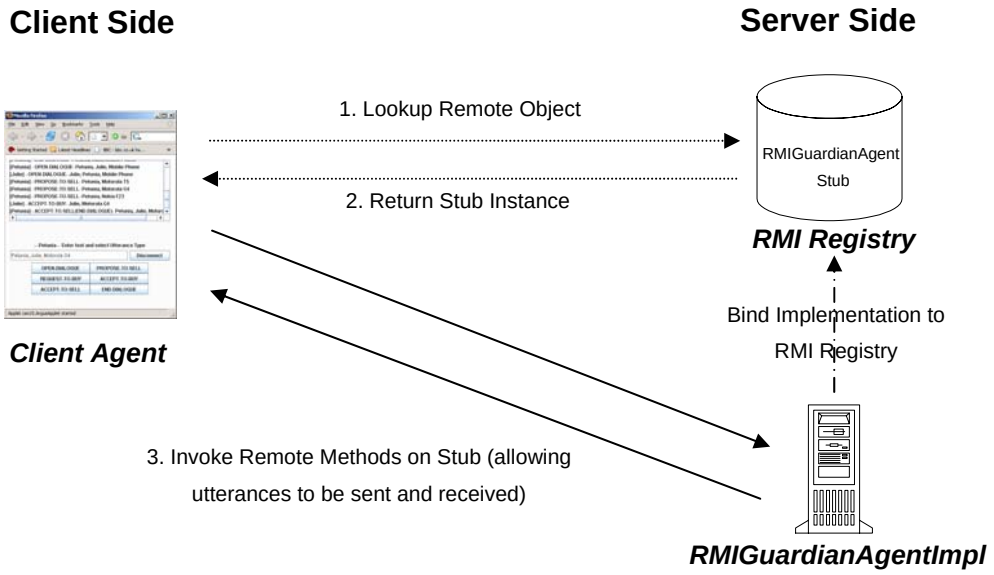


Figure 5.7. Remote clients utilising RMI to connect with the RMIGuardianAgent server.

The two principal methods implemented in the RMIGuardianAgentImpl, sendUtterance and getUtterance, allows a client to push and pull utterances to and from the server. The sendUtterance method tests whether a client has been correctly authenticated and if successful the RMIGuardianAgent stores the utterance allowing additional clients to retrieve the contents when they next poll the server using the getUtterance method. It should be noted that with the current implementation there is a very small probability if two Utterances are sent simultaneously that one Utterance could be lost or the Utterances arrive out of sequence. This is a potentially complex issue and the small probability of occurrence did not warrant further attention in the prototype application.

To increase the functionality provided by RMIGuardianAgentImpl, the object also utilises the Observer design pattern and broadcasts the arrival of a new Utterance by implementing the publisher/subscriber mechanism that is part of the Event Delegation Model available in Java. This allows additional objects to register as an UtteranceListener and receive notification of new utterances being made in a dialogue. Two additional classes were created to support this functionality, the UtteranceListener interface (extending ObjectListener) and the UtteranceEvent (extending ObjectEvent) which encapsulates each new Utterance as the source of the event. Details of two objects that act as UtteranceListeners, providing logging and persistent storage of all utterances within a dialogue, are presented in the following two sections of this chapter.

5.6 LOGGING UTTERANCES

The UtteranceLogBean, created utilising the JavaBeans design pattern (Available: http://java.sun.com/products/javabeans/, 2005) resides on the server and by implementing the UtteranceListener interface receives notification of every utterance made within a dialogue. This enables a continuous log of all the utterances to be maintained and displayed in a server-side Swing-based user interface, as shown in Figure 5.8.

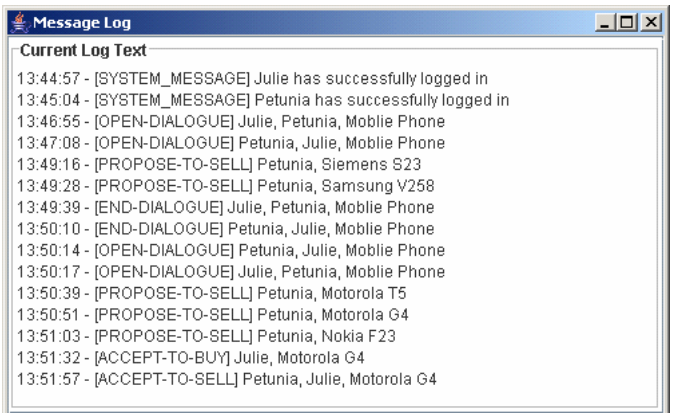


Figure 5.8. The UtteranceLogBean interface.

5.7 STORING UTTERANCES IN PERSISTANT STORAGE

The DataBaseBean class also implements the UtteranceListener interface, but instead of displaying the utterances they are passed to a relational database, implemented using JDBC and the Java-based HSQLDB (Available: http://hsqldb.org/, 2005) database engine, and written to persistent storage. In order to group utterances according to the dialogue in which they were made, the DataBaseBean assigns a unique session ID to each dialogue as it is created (indicated by the OPEN-DIALOGUE utterance). Figure 5.9. shows the entity-relationship diagram of the simple database created.

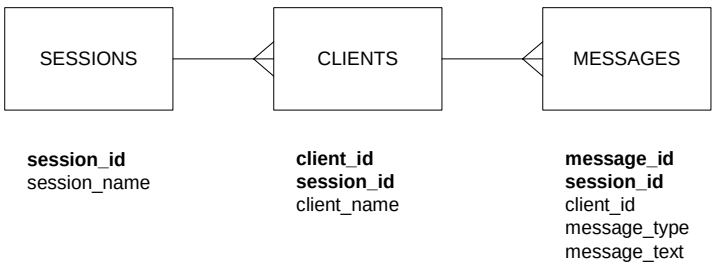


Figure 5.9. ERD diagram of the dialogue database.

To allow a server-side user to view stored dialogues in the database a simple Swing-based GUI is also provided (Figure 5.10.).

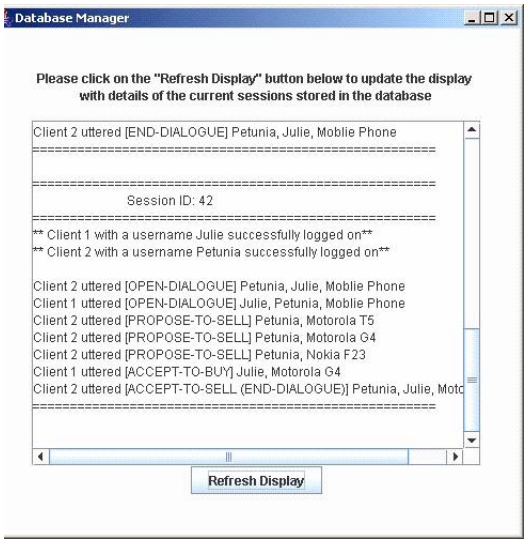


Figure 5.10. The DataBaseBean Interface.

5.8 TESTING

Each component was constantly unit tested during development and the complete prototype application was subjected to thorough integration testing. A demonstration was also provided to a potential user. This section of the chapter outlines the results and demonstrates the correct functionality of the application, showing a complete dialogue game and an example of how inappropriate utterances are prevented from being included in the dialogue.

5.8.1 An Example Dialogue

Figure 5.11. shows a dialogue between two participants using the prototype application presented in this chapter. The dialogue presented here is adapted from the example dialogue given in ASPIC Draft Formal Semantics for Communication, Negotiation and Dispute Resolution document (2005), where Julie, a human user, is interested in purchasing a new mobile phone and therefore enters into a dialogue with Petunia, an agent (in this case also a human user) representing a vendor of mobile phones. Petunia proposes to sell various suitable handsets (PROPOSE-TO-SELL utterances), the first batch of which Julie rejects (by uttering END-DIALOGUE). After several additional proposals, Julie eventually decides to purchase

one (uttering ACCEPT-TO-BUY). The transaction is complete when Petunia confirms she will sell the requested phone (by uttering ACCEPT-TO-SELL).

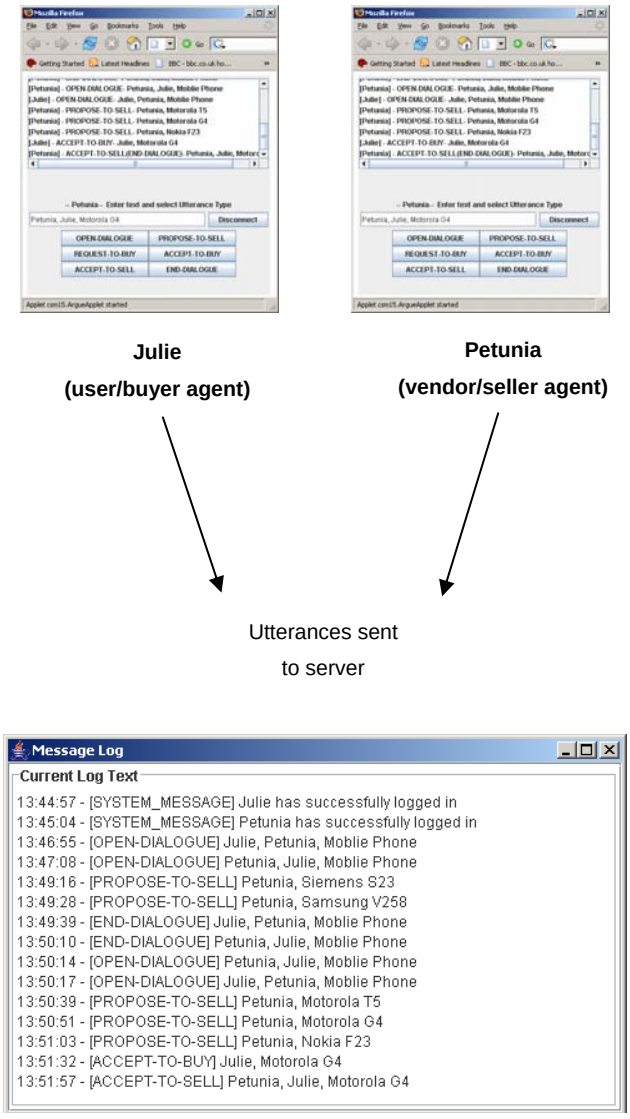


Figure 5.11. Example e-commerce dialogue between Petunia and Julie.

5.8.2 Attempting to make an Inappropriate Utterance

If a user attempts to make an inappropriate utterance in a dialogue, they are prevented and informed accordingly. Extensive testing was conducted at this stage of implementation to check the logic in the semantic enforcement mechanism was correct. Figure 5.12. shows an example of one of these test dialogues in which the PROPOSE-TO-SELL locution has previously been uttered and the current agent is attempting to make a REQUEST-TO-BUY utterance. The semantic preconditions of REQUEST-TO-BUY specify that PROPOSE-TO-SELL must not have been uttered previously and therefore the utterance is not appropriate at this time. Figure 5.12. clearly shows how the utterance is prevented from being included into the current dialogue and the user is informed.

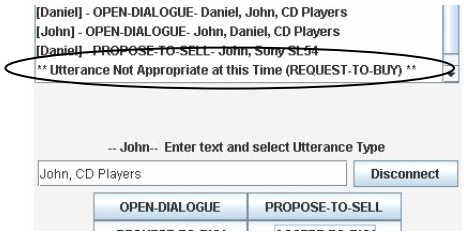


Figure 5.12. An agent may not make an inappropriate utterance at any time.

5.8.3 Outstanding Issues

Several problems were identified with the prototype application that could not be resolved with the current design. The first is that due to the polling mechanism of checking for new utterances there is a small chance (proportional to the polling latency) that if two utterances are made simultaneously then one could be discarded. This would obviously not be acceptable in an application deployed for real-world use, but for this proof-of-concept system the decision was made to not waste resources on part of the system that provided adequate functionality. An additional problem identified was that the client interface would freeze if the RMIGuardianAgent server crashed or was taken offline during an active dialogue. Work will be needed in the next iteration of this application to include a mechanism to inform the user that the server has stopped responding.

5.9 SUMMARY

This chapter has presented the design of the first proof-of-concept prototype dialogue application. The design decisions concerning the deployment location of the enforcement mechanism were discussed, as was the decision to integrate the enforcement into the client user interface components. An overview of the communication mechanism has also been provided and one outstanding problem with the server-side implementation identified. The implementation of the prototype was a success overall, allowing exploration of issues relating to the design of the next set of generic components.

The next generation of the dialogue application will need to be redesigned to be more flexible and the relationship between the user interface, semantic enforcement and the communication mechanism should be loosely coupled. The principle aim of this dissertation is the production of generic enforcement components and therefore the next chapter will focus on augmenting the semantic interface and associated components for this purpose.

Chapter

6 The Generic Enforcement Components

6

Creating the deliberation dialogue and generic enforcement components

6.1 INTRODUCTION

This chapter documents the creation of the deliberation dialogue semantics programmatic representation and the design of the generic enforcement components. The chapter also presents the use of the Template design pattern to create a generic dialogue controller, consisting of three core components, which can be composed with additional applications.

6.2 THE DELIBERATION DIALOGUE OVERVIEW

Hitchcock, McBurney and Parsons (2001) state that a deliberation dialogue focuses on what is to be done in some situation by some agent, either an individual or a group of individuals. A deliberation dialogue differs from other types of dialogue in the absence of fixed initial commitments by any participant on the basic question of the dialogue. The discussion is also a mutual one directed at reaching a joint decision over a course of action and therefore it is necessary that an agent share information and preferences (which may not be beneficial to an agent participating in, for example, a negotiation dialogue). Hitchcock, McBurney and Parsons (2001) discuss that often a deliberation dialogue commences with an open-ended question being posed.

> "A deliberation dialogue arises with a need for an action in some circumstances. In general human discourse, this need may be initially expressed in governing questions which are quite open-ended, as in *Where shall we go for dinner this evening?* Or *How should we respond to the prospect of global warming?*"
>
> (Hitchcock, McBurney and Parsons, 2003: p .5)

A deliberation dialogue also differs from a negotiation or persuasion dialogue in that it is not (at least at the outset) an attempt by one participant to persuade any of the others to agree to

an initially defined proposal. Participants are free to join or leave the dialogue as required and to achieve resolution of a deliberation dialogue, one or more participants must make a proposal for an appropriate course of action.

6.3 SYNTAX AND SEMANTICS OF THE DELIBERATION DIALOGUE

The syntax of the deliberation dialogue, as provided by Hitchcock, McBurney and Parsons (2001) allows multiple agents to participate, which can be human or autonomous entities. However, as stated in Chapter 4, in order to reduce to the complexity of the dialogue application, all the agents participating in a deliberation dialogue will be operated by human users, eliminating the need for a complex decision-making process.

The subject-matter of dialogues can be represented in a prepositional language by lower case Roman letters. Participating agents are denoted by P1, P2, etc and this dialogue also supports a commitment store denoted by CS(Pi) which exists for each agent Pi. This store contains the various propositions which the agent has publicly asserted or preferences he or she has declared. Entries in the store can take two forms, the first being 2-tuples of the form (type, t) where *t* is a valid proposition instance of type *type* with *type* an element of the set {question, goal, constraint, perspective, fact, action, evaluation} and the second being 3-tuples of the form (prefer, a, b) where a and b are proposition actions. Each commitment store is considered to be publicly readable, but only the owner may write to the store. An overview of the syntax for the deliberation negotiation protocol is presented in Figure 6.1, with a comprehensive discussion available in Hitchcock, McBurney and Parsons (2001). This document also contains a public axiomatic semantics for the deliberation dialogue protocol which presents pre-conditions and responses (essentially post-conditions) for the utterance defined in the syntax.

Participants: There can be multiple participants, with participants able to join and leave throughout the course of the dialogue. There must be at least one agent participating in the dialogue at any one time.

Dialogue Goal: A deliberation dialogue focuses on what is to be done in some situation by some agent, either an individual or a group of individuals

Communication Language: The minimum locutions needed for a dialogue between participants in a deliberation dialogue are:
OPEN-DIALOGUE(Pi,q?): Participant Pi proposes the opening of a deliberation dialogue.
ENTER-DIALOGUE(Pj,q?): Participant Pj indicates a willingness to join the deliberation dialogue
PROPOSE(Pi,type,t): Participant Pi proposes proposition t as a valid instance of type type, where type is an element of the set {question, goal, constraint, perspective, fact, action, evaluation}
ASSERT(Pi,type,t): Agent Pi proposes proposition t as a valid instance of type type, where type is an element of the set {question, goal, constraint, perspective, fact, action, evaluation}
PREFER(Pi, a, b) Agent Pi indicates a preference for action-option a over action-option b.
ASK-JUSTIFY(Pj, Pi, type, t) Agent Pj asks agent Pi to provide justification of proposition t of type type, where t is in CS(Pi)
MOVE(Pi, action, t) Agent Pi proposes that each participant(agent) pronounce on whether they assert proposition a as the action to be decided upon by the group.
RETRACT(Pi, locution) Agent Pi expresses a retraction of a previous utterance locution is one of the following three locutions; assert(Pi,type,t) move(Pi,action,a) or prefer(Pi,a,b)
WITHDRAW-DIALOGUE(Pi, q?) Agent Pi announces his/her withdrawal from the deliberation dialogue

Figure 6.1. Overview of syntax for the deliberation dialogue protocol.

MOVE utterance - Syntax MOVE(Pi, action, t)

Meaning: Agent Pi proposes that each participant (agent) pronounce on whether they assert propostion a as the action to be decided upon by the group.

Preconditions: Some participant Pj, possibly Pi, must previously have uttered either propose(Pi,action,a) or assert(Pi,action,a)
Response: None required. Other participants distinct from Pi who wish to support proposition a as the action to be decided upon by the group can respond with the locution assert(Pk,action,a)

Commitment Store Update: The 2-tuple (action,a) is inserted in CS(Pi)

6.2 Example of the deliberation dialogue semantics (MOVE locution).

6.4 CHANGES NEEDED TO THE ENFORCEMENT MECHANISM

The semantic enforcement mechanism developed in the prototype application was adequate for the first generation of components, but with the ultimate aim of this dissertation being for the mechanism to enforce semantics of multiple dialogue types, the existing framework did not provide enough flexibility. The following section of the chapter documents how the existing components were modified to enforce multiple dialogue types.

6.4.1 Modifying the Semantic Enforcement Method Parameters

The e-commerce negotiation dialogue syntax only allowed a maximum of two participants in a single dialogue and therefore the agent at whom an utterance was directed at was implicit. However, the semantics of the deliberation dialogue, and indeed many other dialogue types, require that more than two agents may participate in a dialogue. Therefore the original method of passing an ArrayList of previous utterances for each agent as a parameter to allow preconditions to be checked was no longer practical because passing multiple ArrayList objects, although possible using the Java 5.0 variable argument feature, did not address the critical problem of how to associate each agent with the correct list of previous utterances (see Figure 6.3).

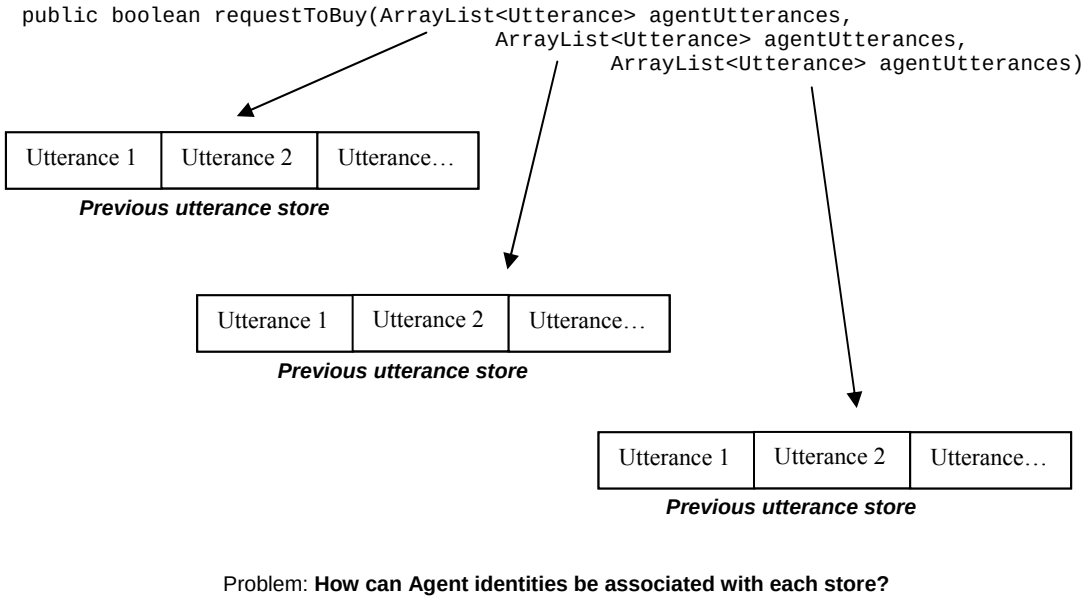


Figure 6.3. Problems with passing multiple ArrayLists of previous utterances.

A new data structure was required that could store utterances and the associated agent identity. The obvious choice was a HashMap data structure, using an agent's identity AgentUser object as a key and the ArrayList of previous of utterances as the associated value (Figure 6.4).

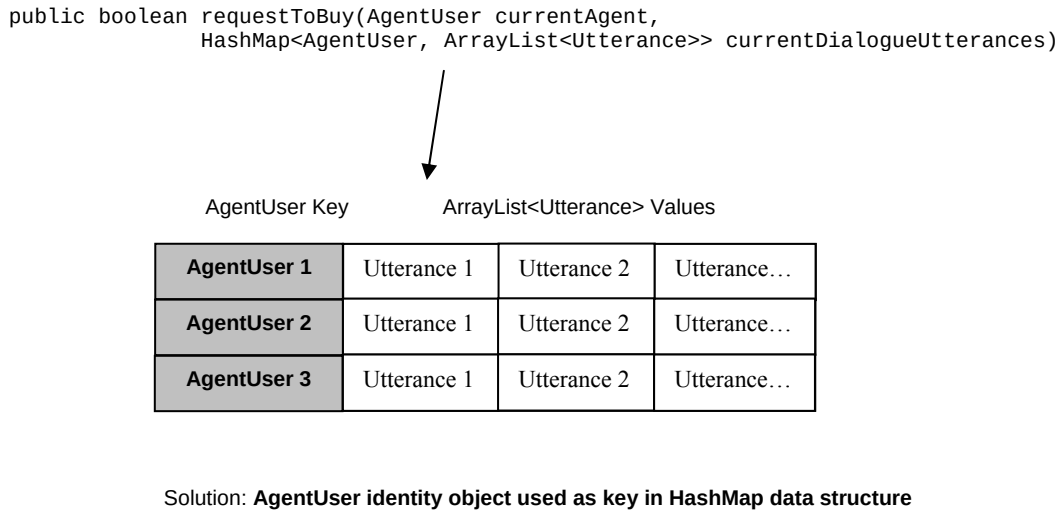


Figure 6.4. The use of HashMaps to store previous utterances.

This change in design lead to a minor modification to the existing AgentUser class where the equals() and hashcode() (inherited from the Object superclass) were overridden. This allowed a particular agents' utterance list to be identified from the collection by generating an AgentUser object with the appropriate details and searching the HashMap for a matching AgentUser key. As multiple agents could be participating in a dialogue this also meant that the current agent attempting to make an utterance must be passed as a parameter into the semantic enforcement mechanism, enabling the associated list of previous utterances made by that agent to be identified.

6.4.2 Modifying the Checking of Previous Utterance Types

Chapter 5 illustrated how the semantic enforcement mechanism was implemented utilising a key method called containsUtteranceType which was capable of determining if a list of previous utterances contained an utterance of a specified type. This method accepted an ArrayList of utterances (either the current agents' or the additional participants') and an integer representation of the utterance as a parameter, returning a Boolean value of true if the ArrayList contained a matching utterance. Although the semantics of the method were valid the parameter list and internal implementation had to be modified to process the new data structure and the potential for an increased number of agents simultaneously participating in a

dialogue. As discussed previously, the preconditions for both dialogue types require that in order for a particular utterance to be appropriate, in addition to the uttering agent having made specific utterances previously (such as OPEN-DIALOGUE), the conditions also often require that another agent has made (or not made) a particular utterance. When only two agents were participating in the dialogue, checking the other agent's previous utterances was simple (the additional list of previous utterances implicitly belonged to the other participating agent and could be passed as a parameter). However, the new semantics allowed multiple agents to participate in a dialogue and establishing all of their identities and checking each list of previous utterances individually would not be efficient. A new design was proposed that consisted of dividing the containsUtteranceType method in two new methods, one capable of checking only the current agent's previous list of utterances and the other capable of checking *all* instances of each of the other participating agent's list of previous utterances. Accordingly two methods were created, named thisAgentUtteredType and otherAgentUtteredType for searching the previous utterances of the current agent and other agents respectively, returning true if a match was found. Figure 6.5. shows an example of the code for the thisAgentUtteredType method.

```
/**
 * Check to see if the current agent has uttered the specified utterance type
 * within the current dialogue
 *
 * @param thisAgent The current Agent's AgentUser
 * @param utteranceType The utterance type
 * @param currentDialogueUtterances HashMap of AgentUser(key) and ArrayList of all
 * Utterances previousley uttered within the current dialogue
 */
public boolean thisAgentUtteredType(AgentUser thisAgent, int utteranceType,
        HashMap<AgentUser, ArrayList<Utterance>> currentDialogueUtterances) {

    //get all of the agents involved in the dialogue
    Set<AgentUser> allAgents = (Set<AgentUser>)currentDialogueUtterances.keySet();

    //loop through all the agents looking for the agent specified
    for (AgentUser agent : allAgents) {
        if (agent.equals(thisAgent)) {
            //found the matching agent (i.e. the current agent)
            ArrayList<Utterance> agentUtterances =
                                        currentDialogueUtterances.get(agent);

            //loop through the agents utterances checking if a match is found
            for (Utterance thisUtterance : agentUtterances){
                if (thisUtterance.getUtteranceType() == utteranceType) {
                    return true;
                }
            }
        } else {
           //found another agents previous utterance store. Ignore as only
           //interested in the current agent
        }
    }
    return false;
}
```

Figure 6.5. thisAgentUtteredType method code adapted to accept HashMap data structure and multiple agents.

6.4.3 Checking Utterance Contents

One of the major changes in the semantics of the deliberation dialogue was the requirement to more closely scrutinise the tuple contents of previous utterances in addition to the type. For example, in the e-commerce semantics the locution tuples only contained a maximum of one element representing a product (or product bundle) being discussed. However, as the deliberation dialogue syntax illustrates, this dialogue allows the locution tuples to be of varying cardinality. The preconditions in the deliberation semantics also require that for an utterance to be appropriate the tuple contents must contain matching elements of subject matter (represented as lower case roman letters). For example, Figure 6.2 in section 6.1.3 of this chapter show the semantics for the MOVE locution in the deliberation dialogue. The preconditions state that in order for the utterance to be appropriate some participant in the dialogue must have previously uttered either propose(Pi,action,a) or assert(Pi,action,a) where *action* and *a* match the contents of the new MOVE utterance. Using an associative matching mechanism, such as contained in the associative blackboard coordination model, this type of matching would be easy to accomplish as the previous utterance tuples could be searched for by simply constructing a template tuple with the required information (and utilising wildcards for unknown information). However, as this system does not utilise such a mechanism, a simplified version of associative tuple matching had to be developed.

The solution developed extracted the tuple contents from each utterance in the list being searched and utilised the Java String indexOf method to search for matching character sequences. The indexOf method returns the starting position in the String (zero indexed) of the matching character sequence, or -1 if the sequence cannot be found. This allowed the creation of two new versions of the thisAgentUtteredType and otherAgentUtteredType methods, named thisAgentUtteredContents and otherAgentUtteredContents respectively, that searched for previous utterances, comparing tuple contents in addition to utterance type. The new methods accept a String in the parameter list that represented the elements of the new tuple that should be matched and utilises the same logic as the previous method, except that when a matching utterance type is found, the utterance contents are also compared to the elements of the new tuple. If the indexOf method returned a value greater than -1 this indicated that the tuple contents of the previous utterance match the new element. Figure 6.6. illustrates how the original e-commerce dialogue semantics ACCEPT-TO-BUY locution was modified to ensure that in addition to the matching the required previous utterance type the contents of the new tuple must contain the same product bundle element as the previous PROPOSE-TO-SELL utterance.

```
      public boolean acceptToBuy(Utterance currentUtterance, AgentUser currentAgent,
HashMap<AgentUser, ArrayList<Utterance>> currentDialogueUtterances,
           HashMap<AgentUser, ArrayList<String>> currentDialogueCommitmentStores){
       if ((thisAgentUtteredType(currentAgent,
EcommerceDialogueSemantics.OPEN_DIALOGUE, currentDialogueUtterances)
       && thisAgentUtteredType(currentAgent,
EcommerceDialogueSemantics.OPEN_DIALOGUE, currentDialogueUtterances))
       && otherAgentUtteredContents(currentAgent,
EcommerceDialogueSemantics.PROPOSE_TO_SELL,
(currentUtterance.getUtteranceText().split(",")[2]), currentDialogueUtterances)) {
           return true;
       }
       return false;
   }
```

Figure 6.6. Modified acceptToBuy concrete implementation of the semantic interface.

Note that the interesting statement in Figure 6.6. is highlighted in bold, where the third element (index position [2]) of the current utterance tuple is extracted using the split method and passed as a parameter for comparison to previous utterance.

6.4.4 Commitment Stores

The semantics of the deliberation dialogue MOVE locution (shown in Figure 6.2) highlight the addition of commitment stores to the dialogue. The contents of the commitment stores indicate dialogical commitments that an agent is willing to defend if challenged by other participants. Preconditions in some of the deliberation dialogue locutions require that an agent's commitment store contain a specific tuple and post conditions specify that tuples must be added, removed or updated to reflect the current state of commitments indicated by the new utterance. This required that commitment stores be treated much like the stores of previous utterances, i.e. they should be publicly viewable meaning all agents should maintain a copy of other agent's commitment stores in their current state. The method developed to store and process previous utterances was easily adapted to perform this function and instead of storing an ArrayList of utterances in a HashMap, an ArrayList of Strings was substituted. There was no benefit to creating an additional class to represent the commitments as they are essentially just tuples of information (only the content of which would need to be checked) and the String split method could easily be used to extract required elements. Accordingly, all the semantic enforcement components were modified to support the newly created HashMap representation of the commitment stores to be passed as a parameter. Figure 6.7. shows an example of the new method signatures.

```
public boolean requestToBuy(Utterance currentUtterance, AgentUser currentAgent,
HashMap<AgentUser, ArrayList<Utterance>> currentDialogueUtterances,
           HashMap<AgentUser, ArrayList<String>> currentDialogueCommitmentStores)
```

Figure 6.7 Example of updated semantic enforcement method signatures.

6.5 CREATING THE GENERIC SEMANTIC ENFORCEMENT COMPONENTS

6.5.1 Overview

Although the semantics enforcement implementation was updated to support the deliberation dialogue it was still tightly coupled with the user interface component. Minsky and Ungureanu (2000) state that a generic control mechanism for heterogeneous distributed system must satisfy the following principles:

> (1) coordination policies need to be enforced (2) the enforcement needs to be decentralised; and (3) coordination policies need to be formulated explicitly – rather than being implicit in the code of the agents involved and they should be enforced by means of a generic, broad spectrum mechanism; and (4) it should be possible to deploy and enforce a policy incrementally
>
> (Minsky and Ungureanu, 2000: p. 1)

The first and second principles have currently been satisfied in this system i.e. the coordination policies (the semantic rules) have been enforced and the enforcement process is distributed to the client-side components. However, the current implementation is violating the third principle, which also prevents the fourth principle from being satisfied. Accordingly, the main focus of this section of the chapter is to document the complete separation of the semantic policy from the enforcement mechanism (also allowing the policy to be deployed incrementally). Although the semantic policy is currently abstracted to the contractual interface, as specified in Chapter 4, the enforcement mechanism also requires a concrete implementation to provide the required functionality (interfaces only provide pure virtual methods) and these concrete methods are currently embedded in the ArgueApplet.

6.5.2 Designing the Dialogue Controller (Semantic Enforcement Mechanism)

The primary design focus for the generic dialogue controller was the need to expose a consistent interface, regardless of dialogue type, to components utilising its functionality. This meant that the semantic interface designed previously (relating to only the e-commerce negotiation dialogue type) needed to be implemented and encapsulated behind another well-defined interface. The dialogue controller component needed to provide three core functions:

1. A generic algorithm to determine if a specified dialogue utterance was appropriate at the given time.
2. Common methods to enable an agent's previous utterances and commitments stores to be managed.
3. Encapsulation of any dialogue-specific functionality

These requirements fitted the specification for the well established Template design pattern (See Freeman and Freeman, 2005 for further discussion on the Template design pattern). Essentially the Template design pattern allows core algorithms and common functionality to be defined in a base class and subclasses to be created that provide specialist behaviour. The template base class specifies two types of methods, the first are hook methods which have a default implementation or are empty, and can be overridden with new behaviour if required. The second types are named slot methods, which are abstract and the required functionality must be implemented in the subclass. The super type provides a consistent interface for the generic algorithms and the common functionality, allowing polymorphic calls on subclasses with customised behaviour.

6.5.3 Utilising the Template Pattern to Create a Generic Controller

An abstract class named DialogueController was created that would act as the template for the generic dialogue controller. For each dialogue that will support semantic enforcement utilising the generic dialogue controller mechanism, a sub-class must be created containing the semantic logic (i.e. this sub-class must also implement the associated semantic interface) and any specialist behaviour. The generic enforcement algorithms in the DialogueController base class were implemented as slot methods and therefore sub-classes have to provide their own implementation. Methods providing functionality such as checking if an utterance is appropriate at the current time must be delegated to the concrete implementation of the semantic interface and therefore they are also specified as slot methods. Common functionality, such as checking if an utterance type has been made by another agent participating in the dialogue, was provided in hook methods (which can be overridden if necessary).

The core methods of the generic super class DialogueController are described below, with full details provided in the Javadoc on the accompanying CD-ROM.

6.5.4 Overview of DialogueController Slot Methods

The slot methods are declared as abstract in the base class and must be implemented in the dialogue-specific sub-class.

utteranceAppropriate

This slot method is responsible for taking an integer representation of an utterance type in the current dialogue as a parameter and returning true if the utterance is appropriate at the given time. Because this method is defined in the base class, this method can be called polymorphically and, as highlighted in Chapter 4, this is where the enumeration function in Java would not provide the required functionality. Due to the inability to define a super type dialogue enumeration, an enumerated representation of an utterance could not be passed as a parameter in a polymorphic call where the dialogue type was not known at compile time. However, an integer representation of the utterance type can take any value representing a locution from any dialogue. If the integer representation is not recognised by the sub-class implementation then an UtteranceNotSupportedException should be thrown back to the calling method. This method essentially encapsulates the logic that was tightly coupled with the utterance type button ActionHandler in the prototype application (documented in Chapter 5 section 5.4.5).

utteranceWellFormed

The new implementation of the enforcement mechanism provides this slot method that should perform any required syntax checking on the utterance tuple contents.

getDialogueType

This method returns the String name of the current dialogue type and is therefore dialogue-dependant and declared as a slot method.

getUtteranceTypeString

This method returns a string representation of an integer utterance type for this dialogue. For example, if the integer value 0 is passed as a parameter to the concrete implementation of the e-commerce dialogue controller, the associated string “OPEN-DIALOGUE” would be returned.

6.5.5 Overview of the DialogueController Hook methods

The following four hook methods are documented in section 6.4.2 and 6.4.3 of this chapter:

ThisAgentUtteredType
OtherAgentUtteredType
ThisAgentUtteredContents
OtherAgentUtteredContents

updateDialogueUtterances
The default implementation of this hook method adds a new utterance, passed as a parameter, to the appropriate agent's ArrayList of previous utterances.

updateDialogueCommitmentStore
The default implementation of this hook method adds a new commitment, passed as a parameter, to the appropriate agent's ArrayList of current dialogical commitments.

6.6 RE-IMPLEMENTING THE E-COMMERCE SEMANTICS

6.6.1 Overview

The production of the generic components necessarily forced the e-commerce semantic enforcement that was previously integrated with the client user interface to be re-implemented. However, the majority of the functionality required to support the enforcement was included in the original prototype and simply had to be extracted and refactored into the new template framework. As Figure 6.9. illustrates, a concrete e-commerce dialogue controller was created by sub-classing the generic DialogueController class (with the keyword extends) and also implementing the contractual semantics interface.

```
public class EcommerceDialogue extends DialogueController implements
                                                    EcommerceDialogueSemantics {
```

Figure 6.9. Class signature for the concrete EcommerceDialogue controller.

The core slot methods were implemented in the concrete EcommerceDialogue class, such as utteranceAppropriate, in addition to all the dialogue-specific utterance methods specified in

the EcommerceDialogueSemantics semantic interface contract. Figure 6.10. shows an excerpt of the implementation provided in the sub-class, illustrating how the logic that determines which semantic utterance checking method to call has been moved from the prototype ArgueApplet class into the generic slot method named utteranceAppropriate.

```
 public boolean utteranceAppropriate(Utterance currentUtterance, AgentUser
currentAgent, HashMap<AgentUser, ArrayList<Utterance>> currentDialogueUtterances,
            HashMap<AgentUser, ArrayList<String>> currentDialogueCommitmentStores)
                                        throws UtteranceNotSupportedException {

        int utteranceType = currentUtterance.getUtteranceType();
        String utteranceContents = currentUtterance.getUtteranceText();

        boolean utteranceAppropriate = false;
        …

        if (utteranceType == EcommerceDialogueSemantics.REQUEST_TO_BUY) {
            utteranceAppropriate = requestToBuy(currentUtterance, currentAgent,
                        currentDialogueUtterances, currentDialogueCommitmentStores);
        } else if {…
        }

        return  utteranceAppropriate;
    }
```

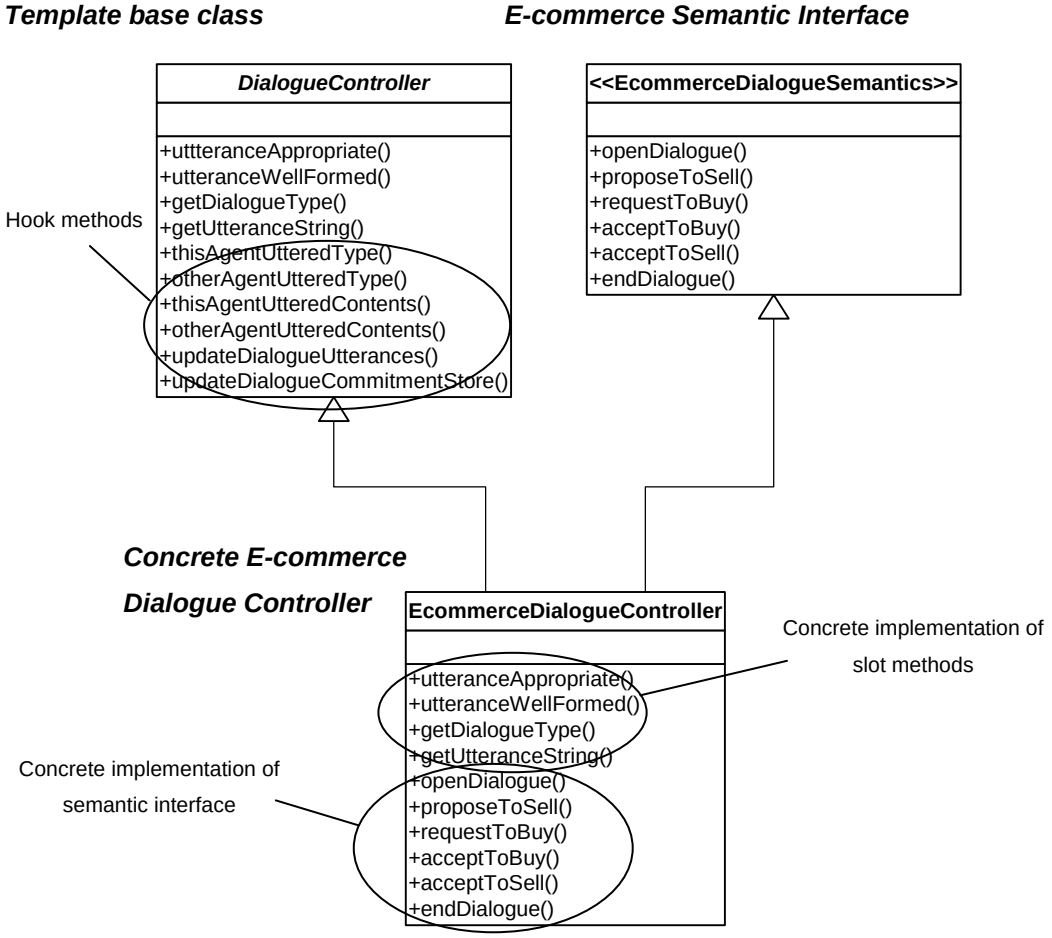
Figure 6.10. Excerpt of the implementation of the utteranceAppropriate slot method.

The final concrete implementation of the e-commerce dialogue controller was also augmented with the new features of the enforcement mechanism. Firstly, the e-commerce dialogue controller now supports simple syntax checking, ensuring that the cardinality of utterance tuples is correct. Secondly, the matching of tuple content was added, which previously had been excluded from the prototype version due to the complexity. Finally, a simple implementation of the commitment stores was also added, with the commitments of buying and selling a product added to the post conditions of the ACCEPT-TO-BUY and ACCEPT-TO-SELL locution implementations respectively.

6.6.2 Annotated Dialogue Controller UML Class Diagram

Figure 6.8. shows an annotated UML class diagram of the EcommerceDialogueController illustrating how the implementation of the DialogueController template and the semantic interface EcmmerceDialogueSemantics creates the dialogue-specific concrete semantic enforcement component. Although the EcommerceDialogueController class is concrete, a variable of this type should never be declared. Instead references to concrete instances should

be loaded into a DialogueController type variable, allowing polymorphic method calling and hiding implementation details (such as the semantic interface methods) from the calling class.



6.8. Annotated UML Class Diagram for the E-commerce Dialogue Controller.

6.7 IMPLEMENTING THE DELIBERATION DIALOGUE SEMANTICS

6.7.1 Creating the Semantics Interface

A semantic interface named DeliberationDialogueSemantics was created, adapting the public axiomatic semantics presented by Hitchcock, McBurney and Parsons (2001) into appropriately named virtual methods (using the technique developed in chapter 4) that can be used to check if a locution's preconditions have been satisfied.

6.7.2 Implementing the Dialogical Commitments

A concrete controller was also created for this dialogue type, sub-classing the generic DialogueController, named DeliberationDialogue which contained both the implementation of slot methods and of the contractual semantic interface. Figure 6.11. shows a code excerpt of the concrete MOVE utterance enforcement method and clearly illustrates the increased number and complexity of associated preconditions that must be implemented when compared with the e-commerce negotiation dialogue (see Figure 5.5. in Chapter 5 for an example). Figure 6.11. also illustrates that if the utterance is determined to be appropriate a dialogical commitment is added to the current utterance object using a new method named setCommitment. This method takes the commitment tuple contents as a String parameter and stores this into a new String variable incorporated into the Utterance class. When the utterance is made, agents can examine the contents of the commitment variable in the Utterance object received, extract any contents and deal with them appropriately.

```
    public boolean move(Utterance currentUtterance, AgentUser currentAgent,
HashMap<AgentUser, ArrayList<Utterance>> currentDialogueUtterances,
            HashMap<AgentUser, ArrayList<String>> currentDialogueCommitmentStores) {
        if ((thisAgentUtteredType(currentAgent,
DeliberationDialogueSemantics.OPEN_DIALOGUE, currentDialogueUtterances)
        || thisAgentUtteredType(currentAgent,
DeliberationDialogueSemantics.ENTER_DIALOGUE, currentDialogueUtterances))
        &&
                ((thisAgentUtteredContents(currentAgent,
DeliberationDialogueSemantics.PROPOSE,
(currentUtterance.getUtteranceText().split(",")[2]), currentDialogueUtterances)
                || otherAgentUtteredContents(currentAgent,
DeliberationDialogueSemantics.PROPOSE,
(currentUtterance.getUtteranceText().split(",")[2]), currentDialogueUtterances))
                || (thisAgentUtteredContents(currentAgent,
DeliberationDialogueSemantics.ASSERTION,
(currentUtterance.getUtteranceText().split(",")[2]), currentDialogueUtterances)
                ||otherAgentUtteredContents(currentAgent,
DeliberationDialogueSemantics.ASSERTION,
(currentUtterance.getUtteranceText().split(",")[2]), currentDialogueUtterances)))) {
            //insert a tuple into the commitment store
            currentUtterance.setCommitment("action," +
                            (currentUtterance.getUtteranceText().split(",")[2]));

            return true;
        }
        return false;
    }
```

6.11. Complex semantic precondition checking in the deliberation dialogue and adding appropriate dialogical commitments.

6.7.3 A Complex Utterance – The Retract Locution

The previous section illustrated how dialogical commitments can be incurred as utterances are made in the dialogue. A new utterance type included in the deliberation dialogue, RETRACT, allows an agent to retract an utterance made previously. Naturally, if an agent indicates that a specific utterance should be retracted then so should the associated dialogical commitment. The original design for the commitment store was based on the design for the previous utterance store which only provides methods to add utterances to the store (there was never the need to remove utterances). However, the contents of the commitment store will change dynamically and must contain only tuples that provide an up-to-date view of the agents' current dialogical commitments. Accordingly, a dialogue-specific method was needed that was capable of removing dialogical commitments from an agent's commitment store. It should be noted that if an utterance is retracted the associated utterance object should not be removed from the previous utterance store, and instead a retract utterance is added. This allows a complete history of utterances to be stored which is required for certain locution preconditions.

The signature of the retractDialogueCommitment method created, accepts the same parameters as the method for updating the commitment store, but instead of simply appending the new commitment to the store, the commitment tuple relating to the retracted utterance is located (utilising the String indexOf method, documented in section 6.4.3) and then removed. Currently the call to the retractDialogueCommitment method takes place in the RETRACT semantic enforcement method, which is not ideal. This means that checking if a RETRACT utterance is appropriate and removal of the associated commitment happens as an atomic action, which could lead to an agents' commitment store not reflecting their true current dialogical commitments. This could occur if an AgentProxy determines a RETRACT utterance is appropriate (causing the current agents commitment stores to be modified), and then the AgentProxy is prevented from sending the utterance to the server. This would lead to the agents' commitment store not being synchronised with the previous utterances made and the agent would have retracted a commitment tuple from its public commitment store without publicly declaring the retraction. Currently, the GuardianAgentProxy checks if the utterance is appropriate and if so, the next statement in the method sends the utterance to the server. Therefore the series of events described could only occur if an Exception was thrown in the GuardianAgentProxy class between checking and sending of the utterance. Due to the small probability of this occurring, this does not warrant further investigation, but if the system were to be deployed in a real-world scenario this behaviour should be modified.

6.8 SUMMARY

This chapter has presented an overview of the deliberation dialogue semantics and the respective modifications required to the semantic enforcement components. The chapter has also presented the design of the generic semantic enforcement component, the dialogue controller, based on the Template design pattern. To enforce the semantic rules of any dialogue type three components are required; the template DialogueController base class, a contractual semantic interface (containing locution types, method signatures for utterance enforcement and information for developers) and a dialogue-specific sub-class of the DialogueController that implements the semantic interface. This set of generic components exposes a well-defined public interface and, as specified as a primary aim of this dissertation, could therefore be implemented in additional applications providing they utilise common dialogue components. The next chapter continues to explain how the dialogue controller components were incorporated into a new more flexible demonstration dialogue application.

Chapter

7 The Complete Dialogue Application

7

Incorporating the generic enforcement mechanism

7.1 INTRODUCTION

This chapter documents the implementation of the final demonstration dialogue application, supporting enforcement for both the e-commerce negotiation and deliberation dialogues. The chapter also contains details of how the client-side of the application was redesigned into a collection of loosely coupled components, encapsulating functionality in order to promote greater extensibility and provide the ability to interchange components.

7.2 NEW COMPONENTS OVERVIEW

In order to produce a loosely coupled set of components allowing the agent, user interface and communication mechanisms to be interchanged, as specified in the aims of this dissertation, the client-side components of the previous prototype application had to undergo a major re-design, with existing classes being refactored and functionality extracted and encapsulated into new components. In addition to the modifications proposed upon completion of the prototype application, the design and production of the new generic semantic enforcement components also affected the implementation of the second generation of application components. Firstly, the deliberation dialogue (and potentially other dialogues) supports more than two concurrent participants, forcing a modification to the way utterances were both stored and passed as parameters (documented in the previous chapter). Secondly, the deliberation dialogue semantics were also more complex, with the preconditions requiring that not only had an agent uttered a particular utterance type, but also that the contents of the tuple uttered contained certain subject element or met specific requirements. Accordingly, syntax checking was needed to enforce that each utterance tuple was of the required cardinality and contained the expected elements. Finally, the new semantics also required the addition of dialogical commitment stores, associated to each participant, which would store an

agent's current dialogical commitment that if challenged would be defended. This meant that the agent component now had to support and manage a commitment store as part of the current state information.

The most logical choice when designing the overall architecture of a client-side agent was to divide the system into a series of components responsible for providing specific functionality. These components needed to be based on a well-defined interface, allowing implementing classes that provided specialist behaviour to be composed together and interchanged freely. The three core abstractions that were implemented consisted of the client user interface (displaying the current dialogue and facilitating user participation), the agent proxy (responsible for managing the agent's state and the communication mechanism) and the dialogue controller (implemented in the previous chapter). Accordingly, the final deliverable dialogue application consisted of the following components:

- A generic semantic enforcement component providing a mechanism to check if preconditions of a specific dialogue utterance have been satisfied (The *DialogueController* abstract class)
- A programmatic representation of two dialogue semantics, presenting preconditions and post-conditions, encapsulated in a combination of a Java Interface (*EcommerceDialogueSemantics and DeliberationDialogueSemantics*) and a implementing concrete class (*EcommerceDialogue and DeliberationDialogue subclassed from DialogueController*)
- A refactored web-based Graphical User Interface (GUI) allowing remote users to connect to an agent proxy in order to participate in the dialogue, provided by the *ArgueApplet* Java Applet.
- A generic agent proxy component encapsulating both the agent's state and the communication mechanism between client and server. An RMI-based GuardianAgentProxy concrete implementation of AgentProxy has been provided
- A server that manages client registration and allows agents to make and obtain utterances within the dialogue. This is provided by the *RMIGuardianAgent* class which permits connection from remote clients using the Java RMI API.
- Logging and server-side display of all utterances made within a dialogue, provided by the *UtteranceLogBean* JavaBean.

These components were grouped into the existing three packages; *client*, *server* and *common* (Appendix II contains full UML class diagrams of the final three packages.) The remainder of this chapter summarises the redesign and implementation of the new components.

7.3 REFACTORING THE USER INTERFACE CLASS (ARGUEAPPLET)

The prototype application had incorporated most of the agent functionality into the client user interface class and although this was suitable for the proof-of-concept application, it did not provide the loose coupling of components required. The first stage in redesigning the client user interface was to identify all code that did not directly provide GUI functionality. As stated previously, it is considered good design practice to code to interfaces instead of implementations and therefore the first new component to be designed was an interface to allow extraction of and support for all the communication and semantic enforcement related functionality needed by the client GUI. The decision was made to name this interface AgentProxy, as a concrete implementation will act as a proxy between the client and server, encapsulating the communication (including the protocols and location of the server) and the semantic enforcement (including the agent's current state). The interface was designed to expose an inherently simple set of methods, allowing the complicated code logic and protocols to be hidden within the proxy. Methods included in this interface allow a client to connect and login to the server, to the make an utterance, (the proxy implementation will be responsible for ensuring that the utterance is appropriate) and retrieve the current state information (now encapsulated within the proxy), such as new utterances and the current commitment store. Accordingly, the new ArgueApplet component only provides user interface functionality and nothing else. Not only is this good object-orientated design, the methods exposed by the AgentProxy interface will also allow new user interfaces to be designed and easily incorporated into the system. For example, an autonomous decision making component that required no human operator could participate in a dialogue by composing the AgentProxy with itself and calling methods as appropriate.

7.3.1 Composing an Agent Proxy

A concrete proxy is composed into the ArgueApplet using an AgentProxy interface type variable to store the concrete implementation. This allows any class implementing the AgentProxy interface to be substituted dynamically and methods called polymorphically. With this methodology the substitution of concrete proxies could potentially even occur at run-time, although care would need to be taken that the current agents' state was not lost. Currently the proxy is instantiated with a concrete class utilising RMI to communicate with the server (discussed further in the next section) although if multiple communication mechanism were available to a client (i.e. the client could specify the communication mechanism used) instantiation could be made more flexible by utilising the Factory design

pattern (See Freeman and Freeman, 2005 for further discussion). Figure 7.1. shows how the reference variable for the proxy was defined in the ArgueApplet component and also how methods are polymorphically called on the concrete implementation without the ArgueApplet being aware of the underlying communication mechanism.

```
private AgentProxy proxy;

// instantiate with concrete implementation, which could be changed both at compile
// time and potentially at run-time
proxy = new GuardianAgentProxy();

proxy.connect(username); //current class not aware of communication method

proxy.getNewUtterance();
proxy.utter(utteranceType, userTypedText.getText());
```

Figure 7.1. Example of polymorphic method calling on AgentProxy interface.

7.3.2 Notifying the Client of New Utterances

The AgentProxy interface provides a method to allow an agent to make an utterance, but the proxy also needs an asynchronous communication method to indicate when an additional participant in the dialogue makes a new utterance. The prototype application highlighted that the continual polling mechanism can be unreliable and therefore in order to keep the interaction between the user interface and proxy loosely coupled the well known Observer design pattern was utilised. Accordingly, the client user interface must implement the UtteranceListener interface and subscribe to the proxy for notification of the new utterances. When the proxy determines a new utterance has been made all registered subscribers are notified by calling the method defined in the UtteranceListener interface. The client can then call a proxy method to obtain the new state information (Figure 7.2.)

```
public interface UtteranceListener extends EventListener {
        public void newUtterance(UtteranceEvent anEvent);
}

public class ArgueApplet extends JApplet implements UtteranceListener {

  // this method is called when a new utterance is received
  public void newUtterance(UtteranceEvent anEvent){

        //get new utterance and state information
        String utterance = proxy.getNewUtterance());
        ArrayList<String> commitmentStore = proxy.getCommitmentStore();
  }
}
```

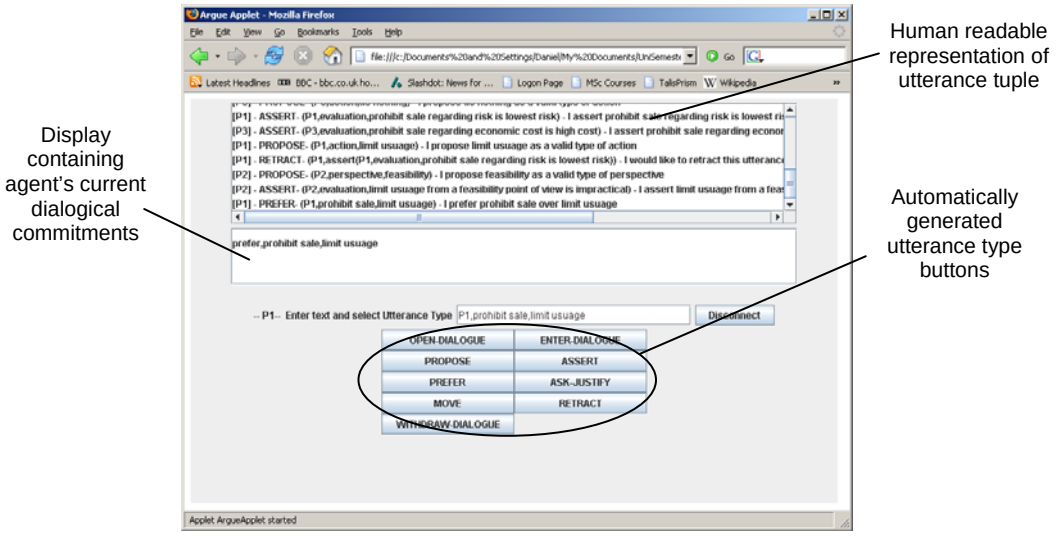
Figure 7.2. Receiving notifications of new utterances.

7.3.3 Generating the Utterance Type Buttons

The previously hard-coded generation of the utterance type buttons was also removed from the ArgueApplet class, as the application will support multiple dialogues. The ArgueApplet class now dynamically generates the buttons by requesting from the proxy a list of utterances supported within the current dialogue. The getUtteranceTypes method included in the AgetnProxy returns an array of Strings containing the names of the utterances (such as "OPEN-DIALOGUE") and the position in the array relates to the integer representation of the utterance type in the semantic interface (e.g. as OPEN-DIALOGUE is the first string in the array indexed at [0], therefore the integer representation of OPEN-DIALOGUE in the e-commerce dialogue is 0). This method could also be utilised to generate other types of user interfaces supporting the choosing of utterance types.

7.3.4 Displaying the Current Commitment Store

The new semantic enforcement components support the use of a commitment store to indicate an agent's current dialogical commitments obtained from previous utterances. Accordingly, the user interface must display an up-to-date representation of the current commitment store. Figure 7.3. shows that an additional text box has been added to the user interface that is responsible for displaying any commitment tuples.



7.3. Screen shot of the new user interface including commitment stores.

7.3.5 Modifying the Utterance Class

When demonstrating the prototype application to potential users, a comment that was frequently made indicated that human users found it difficult to understand the meaning of the predicated tuple format of the utterances. Consequently, in the final implementation of the utterance class an instance variable has been added that can store a String representation of the utterance in a more human-readable format. For example, when an agent utters the tuple "OPEN-DIALOGUE (Daniel, John, Mobile Phones)" the proxy extracts the elements of the tuple and using predefined templates stores the String "Hello *John*, I am *Daniel*. I would like to open a negotiation dialogue with you on the topic of *Mobile Phones*" into the new variable. This can then be displayed to the client (see Figure 7.3).

7.4 DESIGNING A CONCRETE AGENT PROXY

The concept of creating a concrete proxy that facilitated communication between agent and server was adapted from the Aglets Toolkit (Lange and Oshima, 1998), where Aglets are prevented from communicating directly and must communicate through a proxy associated with each agent. This provides added security (preventing public methods of the aglet from being called) and also location transparency. The agent proxy created for this application not only provides location transparency for the server and communicating agents, it also hides the communication mechanism completely. This enables the internal working of the proxies to be modified (for example, implementing security through the use of digital signatures) and providing that the AgentProxy interface is implemented correctly the client will not be affected in anyway.

7.4.1 Encapsulating the Communication Method

Although the use of RMI as a communication mechanism does present several problems in this application (such as the inherent lack of security and reliance on a central server for communication) the decision was made to implement the final application using this method. The prototype demonstrated that RMI provides an efficient way to facilitate communication between agents and a "Guardian Agent" and allows the time available for this dissertation to be focused on more pertinent problems relating to the original research objectives. Accordingly, the relevant client-side methods (and associated instance variables) responsible for RMI-based communication with the server were extracted from the original ArgueApplet

prototype class and placed into a concrete proxy named GuardianAgentProxy. This meant that the application still utilised a client-push/client-pull continual polling method for communication, but this is now encapsulated within the proxy and can be changed easily without affecting other components. For example, Linda-like or peer-to-peer communication could easily be included in a new concrete implementation of the AgentProxy interface without any changes necessary to the client user interface class or semantic enforcement components.

As stated in the previous section, the client user interface is notified of new utterances using the Observer design pattern and must retrieve the new data using appropriate methods within the proxy. Due to this change in the design the new utterances are no longer simply passed to the client interface and displayed which, as noted in the prototype, could lead to messages being delivered out of sequence. They are now placed in a queue (utilising a Java LinkedList), allowing new utterances to be "popped" from the collection in the order they were received. The encapsulation of the communication mechanism also meant that the Utterance class could be hidden from the ArgueApplet user interface component. When a client receives notification of a new utterance, the user interface component calls the getNewUtterance method which returns a simple String representation of the utterance (see Figure 7.4) removing the tight coupling between the ArgueApplet and Utterance classes.

```
public synchronized String getNewUtterance() {
   //remove the first element and return in string format
   Utterance newUtterance = utteranceQueue.removeFirst();
   //convert the utterance to a String to hide Utterance class from client interface
   String utteranceString = "[" + newUtterance.getClientUserName() + "] - " +
               getUtteranceTypeString(newUtterance.getUtteranceType()) +
               "- (" + newUtterance.getUtteranceText() + ") - " +
                                          newUtterance.getUtteranceMeaning();

   return utteranceString;
}
```

Figure 7.4. Converting an utterance to a String representation.

The concrete proxy also encapsulates all the connection and disconnection requests to the server (the client triggers the request using a method exposed in the AgentProxy interface) and is responsible for maintaining the AgentUser client identity obtained from the server (again hiding implementation detail from the client interface, facilitating loose coupling).

7.4.2 Encapsulating the Agent State

In the second generation of application components the proxy is responsible for managing the current state of the agent and relevant information about other participants in the dialogue. Essentially this meant that for each agent participating in the dialogue a list of previous utterances and current commitments should be stored using the HashMap data structures specified in Chapter 6. The enforcement components provide the majority of functionality for managing the HashMap stores and therefore the proxy class required only a concrete implementations of the data structures and appropriate calls to the dialogue controller methods when a new utterance was made or received. Accordingly, as the code in Figure 7.5. illustrates, two HashMap data structure were implemented in the GuardianAgentProxy to represent the complete dialogue state, one storing previous utterances, the other storing current commitments.

```
    private HashMap<AgentUser, ArrayList<Utterance>> currentDialogueUtterances = new
HashMap<AgentUser, ArrayList<Utterance>>();
    private HashMap<AgentUser, ArrayList<String>> currentDialogueCommitmentStores =
new HashMap<AgentUser, ArrayList<String>>();
```

Figure 7.5. Data structures to support storage of previous utterances and current commitments.

7.5 INCORPORATING THE DIALOGUE CONTROLLER

Chapter 6 documented the construction of the generic semantic components (referred to as a dialogue controller) and therefore the concrete GuardianAgentProxy simply had to compose the component into the class and delegate functionality received from the client user interface to the current DialogueController instance. Figure 7.6. illustrates how the concrete implementation of a DeliberationDialogue dialogue controller was loaded into the utteranceController variable utilising the DialogueController base class as the reference type. This allows polymorphic method calls on the concrete controller and therefore the proxy is loosely coupled with the implementation and unaware of the specific dialogue type that is being controlled or enforced.

```
//load concrete deliberation controller into an astract DialogueController variable
private DialogueController utteranceController = new DeliberationDialogue();

//the current class is not aware of the dialogue type being enforced
String[] utteranceTypes = utteranceController.getUtteranceTypes();

utteranceController.utteranceAppropriate(newUtterance, agentUser,
currentDialogueUtterances, currentDialogueCommitmentStores);
```

Figure 7.6. Utilising the DialogueController through composition.

7.6 SUMMARY OF INTERACTION BETWEEN COMPONENTS

7.6.1 Overview

The following section illustrates how the three abstract components specified previously interact to provide the entire suite of client-side agent functionality. Figure 7.7. shows how the three core components are composed together.

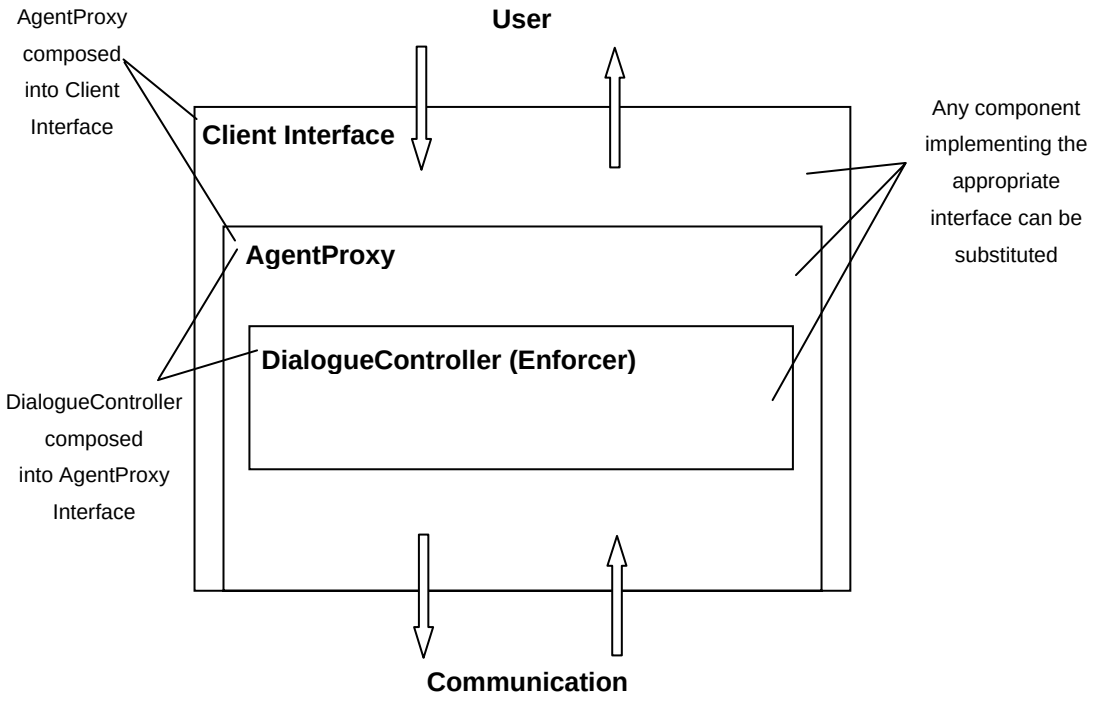


Figure 7.7. Composition of three core components within the client-side components.

7.6.2 Outbound Communication

Figure 7.8. shows how the three components interact when a user makes an utterance in the dialogue. Note that the method calling in the diagrams only utilises the methods defined in the appropriate interface, providing loose coupling between components.

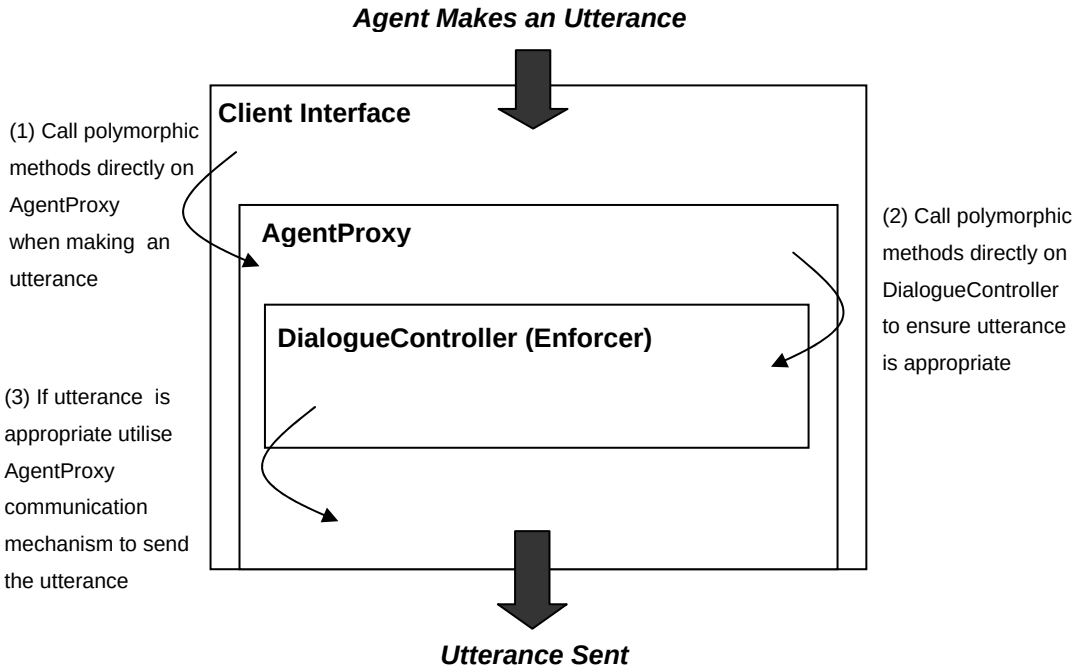


Figure 7.8. Interaction between components when an utterance is made.

7.6.3 Inbound Communication

Figure 7.9. shows how the three components interact when a new utterance is made by another participant in the dialogue. As with outbound communication, only methods exposed as part of the interface are called, with the exception of the client user interface, which utilises the loosely coupled Observer design pattern (requiring initial registration for notification).

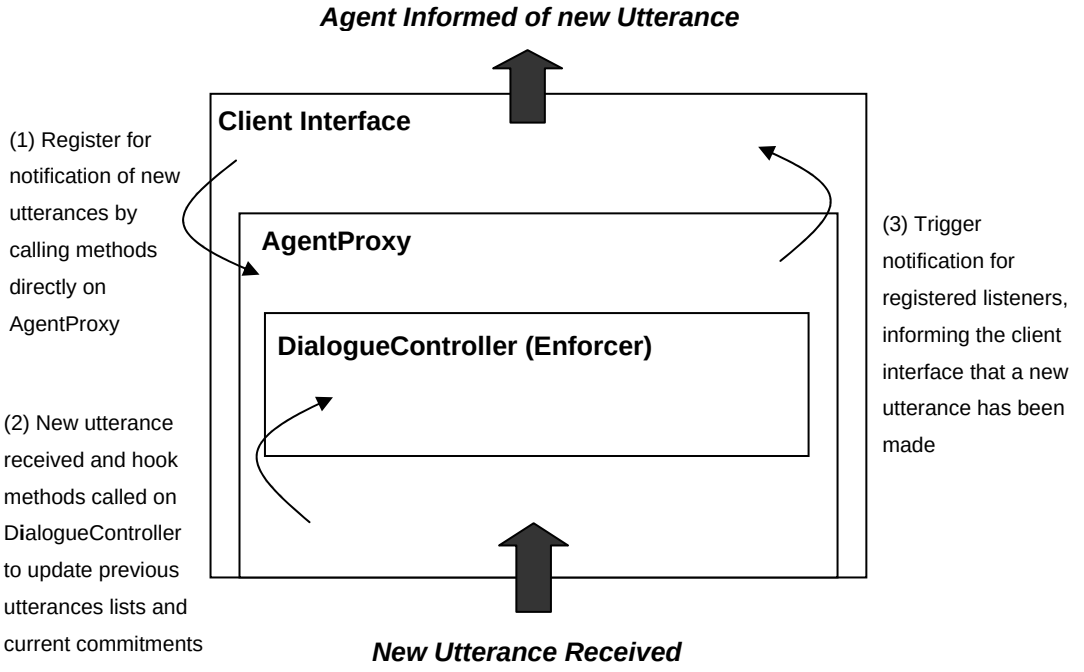


Figure 7.9. Interaction between components when a new utterance is received.

7.7 SERVER SIDE COMPONENTS

The distribution of the semantic enforcement components onto the client-side meant that the majority of work related to creating generic components was conducted there. Accordingly, the existing server-side components developed in the prototype application supported the new client-side components and so the decision was made to keep to RMIGuardianAgent server class which utilised RMI to perform all communications. Only one major modification was made to the server components which related the removal of the DataBaseBean relational database component. Although this bean provided useful functionality in the prototype application, the addition of support for multiple dialogue types meant that the database structure required major modification which was not practical with the timescale provided. This was due to utterance types being representation as integers in the application (to provide type-safety). When the application supported only one dialogue there was a distinct one-to-one mapping for integers to utterance types. However, with the addition of a new dialogue this was not the case. One solution that could be implemented in future work is to associate the dialogue type with each dialogue session in the database and provide a lookup table to determine the relationship between the integer value and the utterance type.

7.8 TESTING

Each component was constantly unit tested during development and the final application was also subjected to thorough integration testing. During the integration testing phase a recurrent problem was identified which lead to the client interface freezing. Utilising the Java Applet console to perform debugging indicated that ConcurrentModificationExceptions were being intermittently thrown when the ArrayList of previous utterances was being accessed. This indicated that the two separate threads operating in the GuardianAgentProxy (one responsible for handling input from the client interface and one for continually polling the server) were both accessing the ArrayList concurrently. The thread responsible for handling input from the client interface was using an Iterator to locate specific utterances in the ArrayList of previous utterance and if the list structure was modified at any time after the Iterator was created (such as adding a new utterance that has been made), the Iterator demonstrated *fail-fast* behaviour and immediately threw a ConcurrentModificationException. The simple solution to this concurrency issue was to synchronize the four methods in the GuardianAgentProxy that modified or accessed the ArrayList by using GuardianAgentProxy as the locking object.

7.8.1 An Example Dialogue

The final dialogue application functionality is demonstrated in this section using a complete execution of a dialogue game that has been adapted from an example given by Hitchcock, McBurney and Parsons (2001). The dialogue consists of three participants, labelled as P1, P2 and P3, who are attempting to decide what action to take regarding the potential health hazards from the use of mobile phones. The dialogue begins with P1 uttering the OPEN-DIALOGUE locution and participants P2 and P3 join the dialogue by uttering ENTER-DIALOGUE. P2 and P3 both propose perspectives from which to consider the question in the dialogue and P1 and P3 state appropriate actions that could be taken. Figure 7.10. show the initial state of the dialogue on the user interface of the P3 agent.

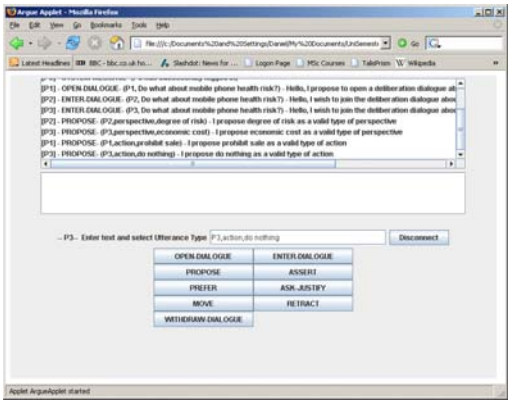


Figure 7.10. Current state of the dialogue on the user interface of the P3 agent.

P1 then asserts that from the perspective of the degree of risk, prohibiting the sale of phones is the lowest risk action-option possible. This creates a dialogical commitment that must be displayed publicly. Figure 7.11. shows a screenshot of agent P1's user interface and the new tuple entered into the dialogical commitment store.

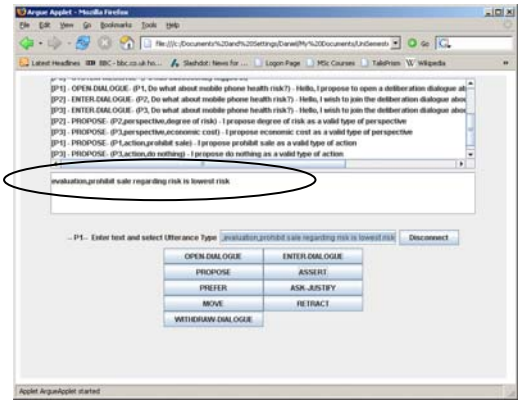


Figure 7.11. New tuple entered into the dialogical commitment store of P1 agent.

P3 asserts a new evaluation, causing P1 to propose a new perspective and retract his previous assertion. Figure 7.12. shows the retraction utterance being made and the effect on the dialogical commitment store.

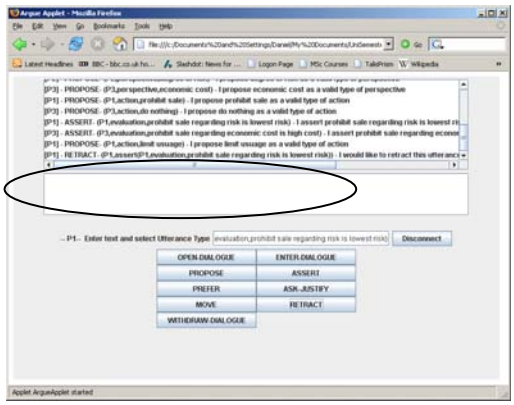


Figure 7.12. Retraction utterance removes the tuple from the dialogical commitment store.

P2 proposes *feasibility* as a new perspective to consider the question and asserts that limiting mobile phone use from the perspective of feasibility is impractical. The dialogue concludes with P1 stating a preference to prohibiting sale instead of limiting usage. Figure 7.13. shows the server-side MessageLog bean output, containing all of the utterances in the completed dialogue.

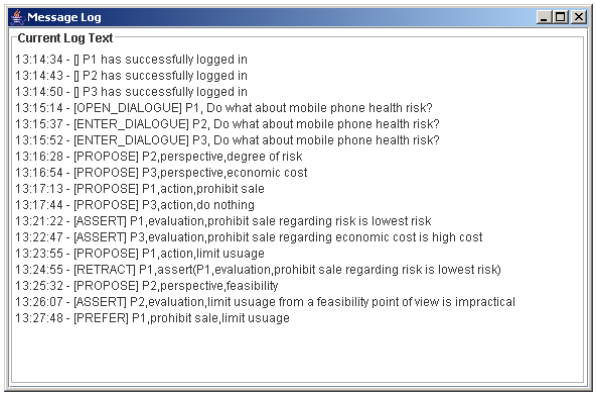
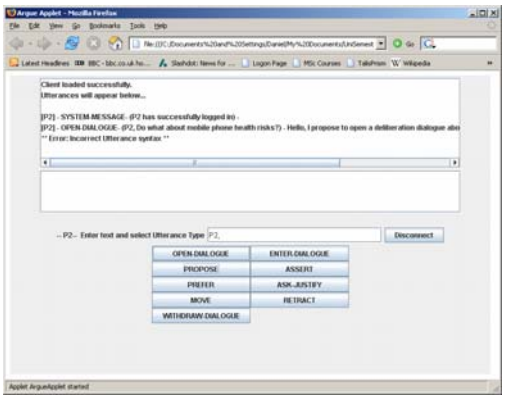


Figure 7.13. Screenshot of the MessageLog bean showing all of the dialogue utterances.

7.8.2 Attempting to Make a Malformed Utterance

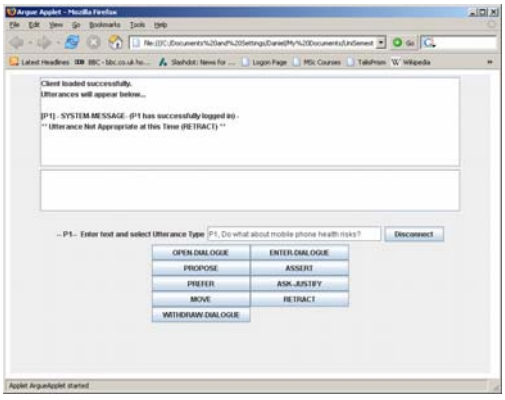
Figure 7.14 demonstrates that if a user attempts to make an utterance and specifies the utterance tuple incorrectly according to the dialogue syntax, they are prevented from making the utterance and informed accordingly.



7.14. The user is informed the utterance syntax is incorrect.

7.8.3 Attempting to Make an Inappropriate Utterance

If at any time a user attempts to make an utterance that would not be appropriate they are prevented from doing so and informed accordingly. Extensive testing of the semantics was conducted to ensure all combinations of preconditions and locutions were enforced. However, due to the limit on available space the screenshots for all tests are not provided. Figure 7.15. shows an example of a user attempting to make an inappropriate utterance.



7.15. The user is informed that the RETRACT utterance is not appropriate at the time.

7.9 SUMMARY

This chapter has presented the design of the final loosely-coupled demonstration dialogue application. Detailed discussion has been provided on the abstraction of the previous client-side agent system into three new components based on well defined interfaces, allowing components to be interchanged freely. The testing conducted in this phase of the project highlighted interesting errors that were mostly caused by concurrency issues, allowing this topic to be explored in more detail. Testing the integrity of the logic within the concrete implementation of the deliberation dialogue was also extremely challenging due to the large combination of dialogue states and preconditions that must be enforced. As the design of final deliverables has been fully documented, the remaining chapters present a summary of and conclusion to the work conducted.

Chapter

8 Discussion

8

Discussion and evaluation of the deliverable components

8.1 INTRODUCTION

This chapter presents both a discussion and evaluation of the deliverable components produced in this dissertation. The chapter begins by evaluating the client-side components, the generic enforcement mechanism and the server-side components of the demonstration dialogue application, and continues by examining issues related to standardisation. The chapter concludes by discussing additional work related to this dissertation.

8.2 CLIENT-SIDE COMPONENTS

Due to the client-side components of the application being composed with the generic semantic enforcement mechanism, the majority of complex design decisions occurred at this location. Both the literature review and comparison of deployment technologies provided an excellent overview of the design of previous agent technologies and facilitated the creation of the supporting architecture for both the agent and enforcement components. The final design of the client-side system consisted of three core abstractions, presenting a well-defined interface for the client user-interface, agent proxy and dialogue controller. This produced a highly extensible system and due to the loose coupling between these components any future extension or modification to the application should not affect the existing components.

The design of the state management in each agent did present several concurrency related problems, such as intermittent exceptions being thrown when multiple processes attempted to simultaneously modify the stored dialogue state in an agent. Although the majority of these problems have been identified and fixed, before the application is deployed in a real-world environment extensive testing should be performed with a large number of agents participating in the dialogue. Existing agent systems, such as Panda (Christian *et al*, 2004),

have also documented similar problems with concurrency and perhaps the risk is inherent due to agent-based systems often operating in very dynamic environments.

Utilising RMI and the direct coordination model for communication between the agents and server provided adequate functionality for both the demonstration application created and the experiments conducted in this dissertation. However, the use of RMI would not scale well in a large scale deployment of a system such as this due to reliance on network stability and a central server. Accordingly, alternative technologies, such as peer-to-peer, should be considered for real-world deployment of this application. The simplicity of implementing the direct coordination model allowed the agents and platform to be developed rapidly and the resources available in this project to be focused on more important areas of research. However, future implementation of an associative blackboard coordination model would provide more functionality for the efficient exchange of information, such as the associative tuple lookup mechanism (which had to be replicated in the dialogue controller components).

8.3 GENERIC ENFORCEMENT COMPONENTS

8.3.1 Overview

The production of the programmatic representations and generic enforcement components was the primary focus of this dissertation, and therefore the majority of the initial deliverable specifications stated in the introduction were satisfied. The use of the Template design pattern to produce the abstract DialogueController class provided an excellent framework for developing and deploying further dialogue type support in this application. The enforcement mechanism could also be utilised in additional applications by integrating the three core enforcement components (the DialogueController, semantic interface and concrete dialogue-specific class) and using methods on the DialogueController interface as a function, returning true if an utterance was considered appropriate given the current dialogue state. Any additional application implementing the components must utilise the method signatures provided in the DialogueController interface and this requires that other components from the application must also be utilised, such as the Utterance class. Consequently, the enforcement mechanism parameters are not truly generic, but the Adapter design pattern could be utilised to provide translation between the interface of the enforcement components and the new application. For any developers wanting to create an Adapter class, they are referred to the Javadoc (specifically the DialogueController and Utterance class) on the accompanying CD-ROM.

In the final version of the dialogue application the dialogue type that will be enforced is hard-coded into the AgentProxy class and, although possible, no method is currently provided to allow the dialogue types to be interchanged at run-time. This functionality could easily be added in the future, with the most appropriate solution being to utilise the Factory design pattern (and therefore encapsulating the code that creates dialogue controllers) to return an instance of a dialogue-specific sub-class of the DialogueController appropriate to the dialogue in which the user wants to participate.

8.3.2 Supporting the Deliberation Dialogue

The deliberation dialogue illustrated that the syntax, semantics and operations supported in different dialogues vary greatly and this provided several new challenges to the implementation of the generic components. Primarily the need for syntax checking was introduced in order to allow successful comparison of tuple contents. The current implementation of syntax checking is somewhat limited as only the correct cardinality of tuples is enforced. However, as the Template design pattern was utilised to create the generic enforcement components the syntax checking method was specified as a hook, allowing new programmatic representations to override existing code and specify more appropriate functionality if required.

The introduction of enforcement for the deliberation dialogue also caused an agent's internal representation of the dialogue state, such as the previous utterances, to be modified to support multiple participants within a dialogue. This can now be considered as a closed issue and the introduction of support for additional dialogue types to support multiple participants would not require further modification. However, the inclusion of the dialogical commitment stores was not specified in any great detail in the deliberation dialogue syntax, essentially only stating that commitments would consists of 2-tuples or 3-tuples. Future dialogue types may place more requirements on the functionality of the commitment store which would have to be implemented in the dialogue-specific sub-class or if this was not possible the generic dialogue controller class would have to be modified. There would also be a strong argument for inclusion into the dialogue controller if more than one dialogue type required this functionality.

Future work on the generic components could also consist of providing default implementations for certain locutions/utterances. Locutions that open and close a dialogue, although named differently in each set of dialogue semantics, essentially perform the same

operation which, as discussed previously, could be extracted from the specialist controller classes to the high-level DialogueController class. Another prime candidate for this would be the retract locution, which currently is implemented as a special case within the DeliberationDialogueController, but would potentially be included in many other dialogue types. Care would need to be taken when designing these generic utterances to make their use as broad as possible, in particular much work would be required on the current implementation of the retract utterance as it is currently tightly coupled with the DeliberationDialogueController. However, this again highlights the potential requirement of redesigning the generic enforcement mechanism each time a new dialogue type is supported. The constant modification of the dialogue controller may prevent a truly generic enforcement mechanism from being created.

8.4 SERVER-SIDE COMPONENTS

The server-side components were implemented mainly to allow the client-side agents to communicate efficiently, and due to the timescale available for this dissertation functionality that would be critical in a real-world server application, such as fault-tolerance and interoperability, was not implemented. However, with only one exception the server-side components did satisfy the specification stated in the introduction of this dissertation. The RMIGuardianAgent component successfully facilitated communication between agents, monitored all the utterances within the dialogue (through the use of the MessageLogBean), but did not allow a server-side user to intervene in the dialogue. This function would be useful in a real-world scenario, and indeed essential in a medico-legal environment where unsafe or illegal actions denoted by utterances must be counteracted. However, the loosely coupled and flexible design of the server components will allow this modification to be conducted and as such, this is suggested for future work.

During the testing of the deliberation dialogue controller it was also observed that the enforcement mechanism did not function correctly if an agent joined a dialogue after it had commenced. This is due to each agent's state in the dialogue being established by recording all the previous utterances made. Consequently, a participant who enters the dialogue after several utterances have been made does not have the complete dialogue history and can therefore make utterances that are not appropriate. Future work could provide a server-side method which provides a joining agent with an up-to-date copy of each agent's previous utterance list.

Evaluation of the server side components also highlighted the inherent absence of security throughout the application. The use of RMI as the communication mechanism meant that all objects are sent unencrypted over the network and potentially client identities and authorisation tokens could easily be intercepted or forged. The obvious solution to this problem is to secure the communication mechanism through the use of cryptographic techniques (for example, utilising SSL). The other inherent security related question, as documented in Chapter 5, is how can the server guarantee the integrity of client-side dialogue controllers? With the current implementation it would be relatively easy to modify the semantic interface or DialogueController class, allowing inappropriate utterances to be made. One potential solution is to implement a second layer of utterance checking at the server-side, but this would defeat the purpose of distributing the enforcement mechanism onto the client. A more efficient solution would be to utilise cryptographic techniques such as digital signatures (for example, include a digitally signed fingerprint of the semantic interface and DialogueController classes in each utterance). As stated previously, the design of the components would support this modification (with the AgentUser object being the most likely mechanism to transport any digital signatures across the network) and this is suggested as a future modification, especially if the agents are to be deployed onto an un-trusted network.

8.5 COMPLIANCE WITH STANDARDS

Chapter 3 highlighted the current efforts to standardise agent technologies and illustrated that the majority of current agent architectures specify only the external interface to an agent platform. Accordingly, internal implementation details were often not specified and as the generic components developed in the dissertation were designed to be deployed internally within an agent, the interface exposed by the dialogue controller did not have to comply with any of the external interface standards.

The current implementation of the agent platform produced in this project, consisting of the agent proxy and server components, was not made compliant to any specific standard due the timescale provided for this dissertation. However, the modular design of the agent and platform will facilitate future work directed at this goal. For example, the agent proxy component is the only interface that would be exposed to an agent platform and therefore a new proxy could be created and deployed that would allow communication with FIPA or MASIF compliant agents.

8.6 RELATED WORK

The core concepts within the LGI model proposed by Minsky and Ungureanu (2000) were utilised extensively throughout the design of the generic dialogue controller and in parallel with work conducted in this dissertation the Moses implementation of the LGI model (Available: http://www.cs.rutgers.edu/moses/) has been enhanced to not only interpret PROLOG rules but also Java-based rules. Consequently, the Moses framework now offers a more generic but less specialised version of the components produced in this dissertation. Future work could combine the generic enforcement components into the Moses framework allowing dialogue-based policies to be enforced in addition to the existing distributed policies.

The generation of the contractual semantic interface provided an effective solution to encapsulate information and operational guidelines for developers of agents that will be using these protocols when participating in dialogue. However, with the current design there is no facility to allow an agent to autonomously interpret the meaning or associated preconditions of each locution within the dialogue. An interesting opportunity for future research is to develop a standard semantics language that could be included in the preconditions and post-conditions of the interface allowing an agent to read the document and interpret the requirements, facilitating autonomous participation in a dialogue without a human operator. Related future work could also automatically generate the semantic interface and enforcement components from a standard semantics language. Potentially the standard semantic language could be implemented utilising a neutral format such as eXtensible Markup Language (XML) to represent dialogue semantics. However, the success of both these proposals are based on the assumption that the enforcement components can be designed to be truly generic, supporting any possible type of locution. This dissertation has already demonstrated that the evolution of the generic enforcement components to support two dialogue types has presented many unexpected problems.

8.7 SUMMARY

This chapter has discussed and evaluated the design and implementation of all the deliverables produced upon completion of this dissertation, and where necessary has suggested future and related work. The next chapter concludes the dissertation, summarising the results of the research and evaluating all of the work conducted.

Chapter

9 Conclusions and Future Work

9

Final conclusions, future work and a dissertation summary

9.1 OVERVIEW

This dissertation has explored the use of argumentation within agent-based communication and coordination, investigating two dialogue game scenarios where two or more agents are attempting to reach a predetermined goal through discourse. The dissertation has illustrated how a series of public axiomatic semantics presented for the dialogues (defining conditions under which an utterance is considered appropriate) have been transformed in a Java-based programmatic representation, created using a semantic interface and implementing concrete class, and enforced by a generic dialogue controller mechanism. A fully functional agent-based distributed dialogue game application has also been presented that demonstrates the capabilities of the generic semantic enforcement components. At the client-side the semantic interface and enforcement components are combined to create a dialogue controller which enforces that at any given time in a dialogue an agent can only make an utterance that is appropriate, i.e. the utterance furthers the desired outcome of the interaction. The server-side implementation of the application acts as a "Guardian Agent", facilitating communication between agents and monitoring the dialogue. The remainder of this chapter illustrates both successes and problems with the research conducted, suggests future work and concludes with a final evaluation of the work undertaken in this dissertation.

9.2 CONCLUSIONS AND FUTURE WORK

The deliverables produced upon completion of this dissertation satisfied the initial requirements of creating programmatic representations of the e-commerce negotiation and deliberation dialogues and producing a re-usable generic enforcement mechanism. Extensive testing of the agent-based dialogue application produced indicated that participating agents where restricted to only making appropriate utterances throughout the dialogue.

The final design of the client-side application consisted of three core abstractions, presenting a well-defined interface for the client user-interface, agent proxy and dialogue controller. This produced a highly extensible system and due to the loose coupling between these components any future extension or modification to the application should not affect the existing components. Several problems with the choice of using the direct coordination model and RMI-based communication were identified, and suggestions for improvement include incorporating a new agent proxy that supports a Linda-like implementation, such as JavaSpaces (Sun Microsystems, 2005) or ideally Panda (Christian *et al*, 2004) which utilises peer-to-peer networks and would also remove the reliance on a centralised server for communication.

The use of the Template design pattern to produce the generic dialogue controller class provided an excellent framework for developing and deploying further dialogue type enforcement using these components. The Template pattern allowed a common protocol to be developed for the dialogue-specific controllers, and in addition to providing default enforcement behaviour, methods may also be overridden to provide extra functionality if required. In the final application the choice of dialogue type controller is hard-coded into the agent proxy and although the design allows controllers to be interchanged at run-time, currently no method to facilitate this is provided. This could easily be added in the future by utilising the Factory design pattern (and therefore encapsulating the code that creates dialogue controllers) to instantiate a dialogue-specific sub-class of the generic dialogue controller according to the agents' dialogical requirements.

The server-side components, although providing adequate functionality for the experiment conducted, highlighted the inherent absence of security in the application which is a common problem with agent technology. The main security concern connected to this research is how the integrity of a client-side dialogue controller can be guaranteed by the server components. One possible solution is to utilise cryptographic techniques such as digital signatures (for example, including a digitally signed fingerprint of the semantic interface and DialogueController classes in each utterance). If the technology developed in this dissertation was to be deployed onto an un-trusted network where the integrity of the client cannot be guaranteed, high priority should be given to the implementing these modifications.

Relating this work to next-generation computational services that will require the use of agents capable of effectively engaging in negotiation and deliberation, the semantic interfaces produced here provide an efficient solution to encapsulating information for the developers of agents that will be using these protocols. However, services such as the Semantic Grid (De

Roure, Jennings and Shadbolt, 2001) and the Digital Business Ecosystem project (Di Corinto and Rathbone, 2004) will require the use of autonomous agents that are capable of dynamically supporting and engaging in multiple dialogue types. Future work should therefore focus on the development of a standard representation of semantics which could be incorporated into the programmatic representation, allowing an agent to autonomously interpret the supported locutions and associated preconditions. However, the investigation conducted in this dissertation of just two different dialogue types has illustrated that due to the syntax, semantics and operations supported in each dialogue varying greatly, it may be extremely challenging to create a truly generic enforcement mechanism.

9.3 FINAL SUMMARY

Undertaking the research documented in this dissertation was extremely challenging and very rewarding. Many new software-engineering techniques, technologies and frameworks were discovered and learnt, thereby augmenting existing skills. Identifying and learning about agent-based coordination models was very interesting and illustrated the challenge of designing large software systems consisting of many dynamic interacting components. Throughout this dissertation the requirement for next-generation agent systems to support more flexible and powerful methods to exchange and evaluate information was constantly encountered, and is something that will most likely be investigated in the computing research community for many years.

When developing the deliverable components extensive research was conducted on software-engineering design patterns, highlighting the need to design flexible loosely coupled components that expose a well-defined interface. As soon as this methodology was put into practise the benefits quickly became apparent, allowing components to be redesigned and refactored with minimal impact of the rest of the application. Testing of the dialogue application also proved extremely challenging due to the large number of agent states and precondition combinations that had to be satisfied. Accordingly, new techniques for the production of a comprehensive and thorough testing strategy were identified and learnt. In conclusion, the production of this dissertation and associated deliverables has greatly added to the author's existing knowledge, abilities and skills and has encouraged a keen interest for research within the computing domain that will continue to be developed.

Chapter

10 References

A complete list of all referenced work

10

ASPIC Consortium (2005) *Draft Formal Semantics for Communication, Negotiation and Dispute Resolution*. Classified Technical report.

ASPIC Consortium (2004) *Theoretical framework for Argumentation*. Classified Technical report.

Berners-Lee, T., Hendler, J. and Lassila, O (2001) The Semantic Web. *Scientific America*. Available: http://www.sciam.com/article.cfm?articleID=00048144-10D2-1C70-84A9809EC588EF21 [Accessed: 10/02/05]

Bellifemine, F., Poggi, A. and Rimassa, G(1999) JADE – A FIPA-compliant agent framework. *Proceedings of PAAM'99*, 97-108.

Bratman, M. E. (1990). What is intention? In Cohen, P. R., Morgan, J. L., and Pollack, M. E., editors, *Intentions in Communication*, The MIT Press. 15-32.

Cabri, G., Leonardi, L. and Zambonelli, F (1998) Reactive tuple spaces for mobile agent coordination, *In Lecture Notes in Computer Science*, v 1477, 237.

Cabri, G., Leonardi, L. and Zambonelli, F (2000) Mobile-agent coordination models for Internet applications. *IEEE Computer*, 33(2), 82-89

Carriero, N. and Gelernter, D. (1989) Linda in Context. *Comms. of the ACM*, 32(4), 444-458.

Christian, A., Duarte, M., Nielson, S., Pound A. and Sandler, D (2004) *Panda: An Implementation of Generative Communication on a Structured Peer-to-Peer Overlay.* Available: http://www-ece.rice.edu/~duarte/images/elec520final.pdf [Accessed: 06/06/05]

Ciancarini, P.; Tolksdorf, R.; Vitali, F.; Rossi, D.; Knoche, A. (1997) Redesigning the Web: from passive pages to coordinated agents in PageSpaces, *In Proceedings Third International Symposium on Autonomous Decentralized Systems*, 377 – 384

De Roure, D., Jennings, N. R. and Shadbolt, R. (2001) *The Semantic Grid: A Future e-Science Infrastructure.* Available: http://www.semanticgrid.org/documents/semgrid-journal/semgrid-journal.pdf [Accessed: 20/02/05]

Di Corinto, A. and Rathbone, N (2004) *Digital Business Ecosystems: The Internet's new Common Land.* Available: http://www.digital-ecosystem.org [Accessed: 10/06/05]

Foundation for Intelligent Physical Agents (2005) *Welcome to FIPA!* Available: http://www.fipa.org/ [Accessed: 11/02/05]

Franklin, S. and Graesser, A. (1997) Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents. *Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages,* Springer-Verlag.

Freeman, E., Hupter, S. and Arnold, K. (1999) *JavaSpaces Principles, Patterns and Practices*. Addison-Wesley, USA.

Freeman, E. and Freeman, E. (2004) *Head First Design Patterns,* O'Reilly Media, USA.

Gelernter, D and Carriero, N. (1992) Coordination languages and their significance. *Communications of the ACM.* 35(2), 97 - 107

Genesereth, M. R. and Ketchpel, S. P. (1994). Software agents. *Communications of the ACM*, 37(7), 48-53.

Genesereth, M. R and Nilsson, N. (1987) *Logical Foundations of Artificial Intelligence*. Morgan Kaufmann, USA.

Georgeff, M. P., Pell, B., Pollack, M., Tambe, M. and Wooldridge, M. (1999) The belief-desire-intention model of agency. *In Intelligent Agents, V,* LNAI Volume 1555, Springer, Berlin, 1-10.

Hamblin, C. L. (1970) *Falacies.* Methuen, London, UK

Hitchcock, D., McBurney, P and Parsons, S. (2001) *A Framework for Deliberation Dialogues*. Available: http://www.humanities.mcmaster.ca/~hitchckd/deliberationdialogues.pdf [Accessed: 11/04/05]

Kotz, D. and Gray, R. S. (1999) Mobile Agents and the Future of the Internet, *In ACM Operating Systems Review*, 33(3), 7-13.

Koukoumpetsos, K. and Antonopoulos, N. (2002), Mobility Patterns: An Alternative Approach to Mobility Management, *Proceedings of the 6th World Multi-Conference on Systemics, Cybernetics and Informatics* (SCI2002).

Lange, D. B. and Oshima, M. (1998) *Programming and Deploying Java Mobile Agents with Aglets.* USA, Addison Wesley Longman, Inc.

Lange, D. B. and Oshima, M. (1999) Seven Good Reasons for Mobile Agents, *In Communications of the ACM*, 42(3), 88-89.

Man Machine Systems (2004) *Design by Contract for Java Using JMSAssert*, Available: http://www.mmsindia.com/DBCForJava.html [Accessed: 08/05/05].

Meyer, B. (2000) *Object-oriented Software Construction*, Prentice-Hall, USA.

Milojicic, D. S., Oshima, M., Tham, C., Virdhagriswaran, S., White, J., Breugst, Markus., Busse, I., Campbell, J., Covaci, S., Friedman, B., Kosaka, K., Lange, D. B. and Ono, K. (1998) MASIF: The OMG Mobile Agent System Interoperability Facility. *Proceedings of the Second International Workshop on Mobile Agents*, 50 – 67

Minsky, N. H. and Ungureanu, V. (2000) Law-governed interaction: A coordination and control mechanism for heterogeneous distributed systems, *In ACM Transactions on Software Engineering and Methodology*, 9(3), 273-305.

Object Management Group (2005) *OMG Agent Platform Special Interest Group*, Available: http://www.objs.com/agent/ [Accessed 11/02/05]
Omicini, A. (1999) Tuple centres for the coordination of Internet agents, *In Proceedings of the 1999 ACM symposium on Applied computing*, 183 – 190.

Oxford University Press (2005) *Oxford English Dictionary*. Available: http://www.oed.com [Accessed: 06/06/05]

Papadopoulos, G. A. and Arbab, F (1998) Coordination Models and Languages, *Advances in Computers,* Academic Press, Orlando, 329-400.

Sierra, K. and Bates, B. (2003), *Sun Certified Programmer & Developer for Java 2*, McGraw-Hill/Osborne, California USA.

Sun Microsystems (2005) *Java Technology*, Available: http://java.sun.com/j2se/index.jsp [Accessed 07/02/05]

Sun Microsystems (2005) *Java Remote Method Invocation Technology,* Available: http://java.sun.com/products/jdk/rmi/ [Accessed 07/02/05]

Sun Microsystems (2005) *Getting started with JavaSpaces*, Available: http://java.sun.com/developer/technicalArticles/tools/JavaSpaces/ [Accessed 07/02/05]

Sun Microsystems (2005) *Using Assertions in Java Technology,* Available: http://java.sun.com/developer/technicalArticles/JavaLP/assertions/ [Accessed: 08/02/05]

Sun Microsystems (2005) *Javadoc Tool Home Page,* Available: http://java.sun.com/j2se/javadoc/ [Accessed: 08/05/05].

Sun Microsystems (2005) *JavaBeans,* Available: http://java.sun.com/products/javabeans/ [Accessed: 08/05/05].

The hsqldb Development Group (2005) *hsqldb - 100% Java Database,* Available: http://hsqldb.sourceforge.net/ [Accessed: 08/05/05].

Toulmin, S. E. (1958) *The Uses of Argument*. London, Cambridge University Press.

Walton, D. N. and Krabbe, E. C.W. (1995): *Commitment in Dialogue: Basic Concepts of Interpersonal Reasoning*. USA, State University of New York Press.

Wooldridge, M. (2002) *An Introduction to MultiAgent Systems.* Chichester, John Wiley & Sons Ltd.

11 Appendix I - Prototype Application UML Diagrams

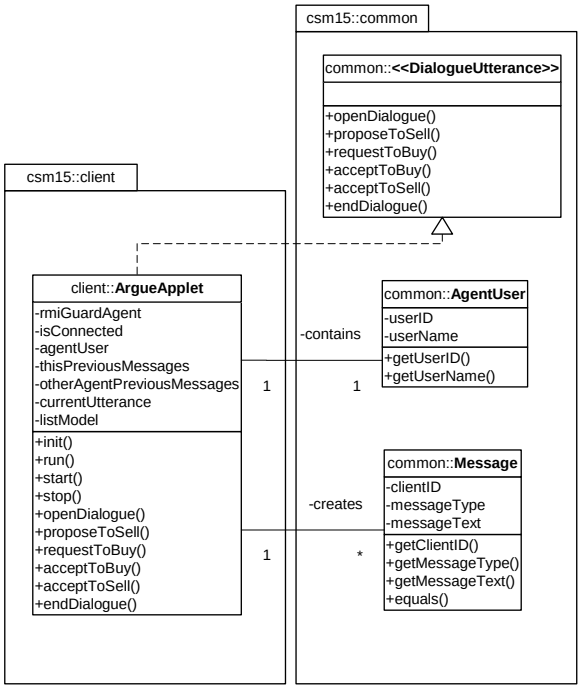


Figure AI.1. Prototype 'client' and 'common' package class diagram

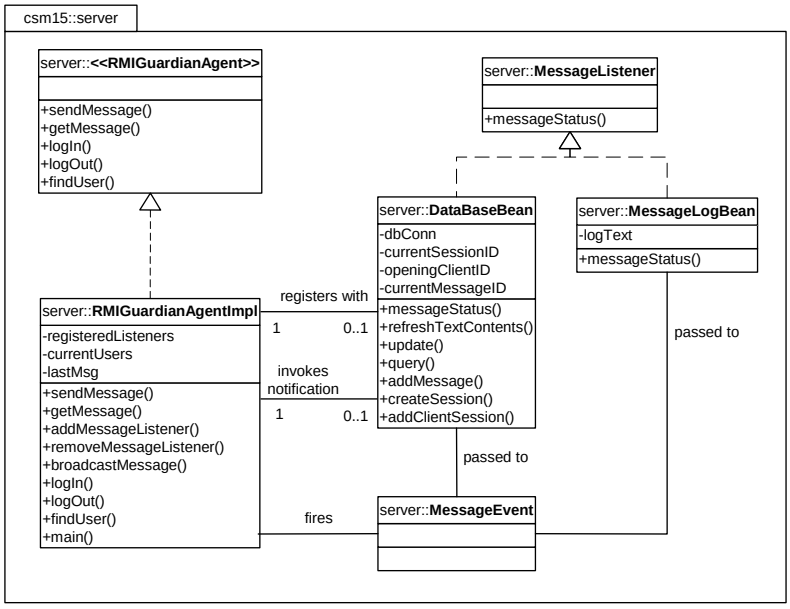


Figure AI.2. Prototype 'server' package class diagram (also showing MessageEvent model)

12 Appendix II - Final Application UML Diagrams

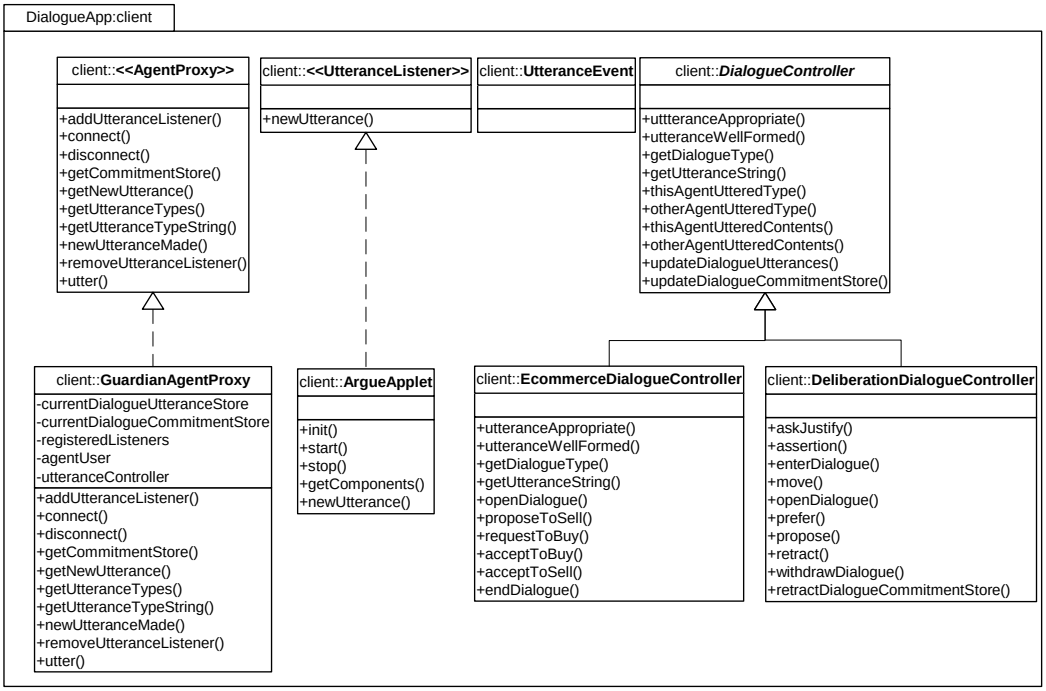


Figure AII.1. Final 'client' package class diagram.

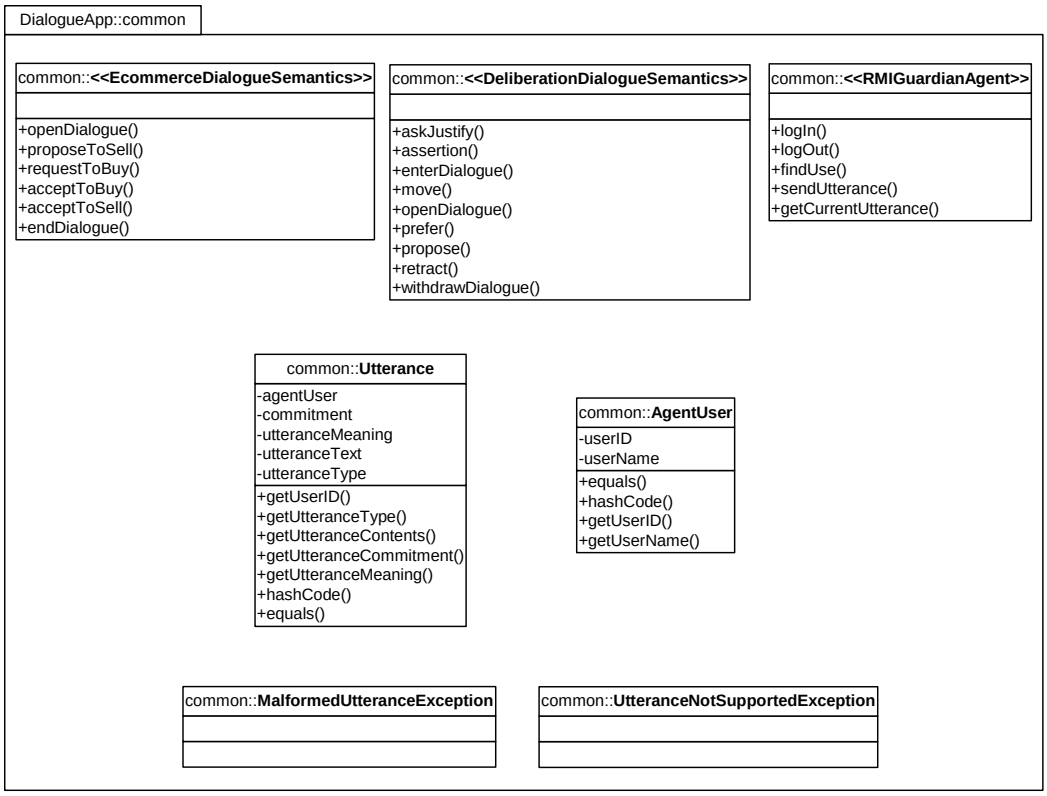


Figure AII.2. Final 'common' package class diagram.

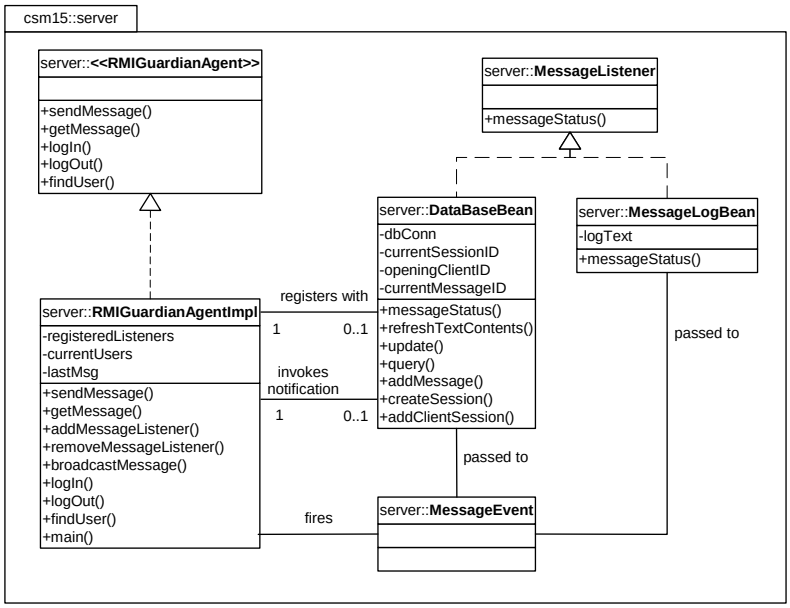


Figure AII.3. Final 'server' package class diagram (also showing MessageEvent model)

13 Appendix III – CD-ROM Contents

The CD-ROM attached to this dissertation contains the following files and directories:

csm1db-finaldissertation.doc

An electronic copy of the full dissertation submitted (this document).

src (directory)

Complete Java source files for all of the application components.

javadoc (directory)

Full Javadoc API documentation for all of the application components. (Click index.html to display in a web browser)

demonstration (directory)

Compiled and executable version of the agent-based distributed dialogue game application. *Note: This application requires that Java J2SE (minimum version 1.4) is installed and functioning correctly (with a path to the RMI Registry defined and the classpath variable set appropriately)* To execute the demonstration please click "Deploy RMIGuardianAgent.bat" and wait until the service has been registered with the RMI Registry (confirmation will appear in the command window). The "ArgueAppletLoader.html" file can be loaded into a Java-compliant web browser to create a user-interface agent that will connect to the RMIGuardianAgent and allow participation within a dialogue.

build (directory)

Compiled version of the final application components sourcecode.

README.txt

Text file containing the contents of the CD-ROM.