Automating Internet Auctions with Adaptable Mobile Agents

Mark Seymour

A thesis submitted for the degree of

Doctor of Philosophy in Computer Science

University of London

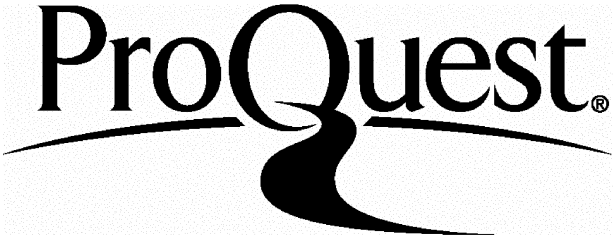
Department of Computer Science

University College London

Gower Street

London WC1E 6BT

February 1999



Abstract

This thesis investigates the automation of different auction methods using computer technology. There are many auction methods. However, four methods are considered for automation. They are First Price Sealed-bid (FPSB), Vickrey, English (e.g. antique and cattle auctions) and Dutch (e.g. flower and tobacco auctions). Since these auctions have well defined bidding and selling rules, it is feasible that the whole process of bidding and selling could be automated. In particular, this thesis investigates automating these auctions using software agents and the Internet.

The main aim of this thesis was to create an **Internet Auction System (IAS)** where users can auction items on their own computer and find and bid in remote auctions held on other users' computers. Adaptable (Seller) agents are used to automate selling and **Adaptable Mobile** (Bidder) **Agents (AMA)** are used to find these remote auctions and to automate the bidding process once they arrive there. The actual items auctioned are not important, though it is intended that agents can only sell or bid for one item at a time.

Developing the IAS is important for two reasons. Firstly, by demonstrating that AMA(s) can automate decentralised auctions illustrates how they could make effective tools to automate other electronic commerce and trading systems. Secondly, by effectively embedding intelligence into agents, users would have greater confidence in allowing agents to operate in these markets on their behalf.

There are four parts to this thesis: (i) the investigation of existing agent-based auction systems; (ii) the development of AMA(s); (iii) the utilisation of these agents to build an IAS and (iv) an evaluation of the IAS using AMA(s).

Researching existing agent-based auction systems highlighted the lack of fully automated auction systems. Most Internet or Web-based auctions do not use agents. They require users to submit bids via e-mail or web page forms. Fewer still give users any control over where or when the auction is conducted or what auction method is used. Therefore, in this thesis agent scripts were implemented as finite automata that were both mobile over the Internet and self-modifiable. Within each script, the agent's intelligence was encoded using fuzzy-genetic strategies. This enabled users to evolve their agents in an auction simulator that used **Fuzzy-Genetic Algorithms (FGA)**. These tools and concepts were then used to design the IAS.

The IAS was assessed by how well the AMA(s) automated various auction methods and by how well the auction simulator evolved rational bidding and selling strategies.

Three specific research contributions were made to agent-based auction systems. Firstly, novel AMA(s) were developed that used self-modifying SGML-based scripts with embedded fuzzy-genetic adaptable strategies. Secondly, AMA(s) were successfully used to automate various auction methods over the Internet. Thirdly, an auction simulator was developed that enabled users to evolve profitable (and in many cases rational) bidding and selling strategies for their respective Bidder and Seller agents to use in the IAS auctions.

Dedication

For Jasmine, my parents, and the teachers of The Nobel School.

Acknowledgements

I would like to thank Professor Philip Treleaven for his tireless enthusiasm and moral support. His optimism, humour, and anecdotes provided a welcome change from the routine of writing up. In addition, I would like to thank Adrian Wilkinson for his time spent proof reading, and his suggestions and comments.

This thesis would not have been possible without the financial support of Xerox Corporation who sponsored the thesis throughout the first year and a half. Moreover, the thesis would not have been completed without the generous funding of NCR who sponsored the last year and a half. I am grateful for their foresight and encouragement.

List of Figures and Tables

Figure 1.1 Thesis areas and overview

Figure 1.2 A user's computer connected to the IAS to form an auction site

Figure 2.1 Electronic commerce research areas

Figure 2.2 Electronic negotiation research areas

Figure 3.1 Computer running multiple agent client/servers

Figure 3.2 Two computers each running one agent client/server

Figure 3.3 Agent moving from one user to another

Figure 3.4 User log-in outcomes

Figure 3.5 Agent filtering

Figure 3.6 An agent network with a Proxy server

Figure 3.7 A simple GUI for a mobile agent system

Figure 4.1 The Adaptable Mobile Agent generic structure

Figure 4.2 Mobile agent state transition diagram

Figure 4.3 SGML as a meta-language

Figure 4.4 An example of an agent script using tags

Figure 4.5 Pseudo code for an agent interpreter

Figure 4.6 Fuzzy set partitioning

Figure 4.7 Boolean and numeric decision chromosomes

Figure 5.1 The IAS consisting of user auction sites acting as bidders and/or sellers

Figure 5.2 Routing agents

Figure 5.3 Application software components

Figure 5.4 The Internet Auction System GUI

Figure 6.1 The Seller agent

Figure 6.2 Utilising the Seller agent and its implementation

Figure 6.3 Auction state space

Figure 6.4 Fuzzy set partitioning of 'expected bidder valuation' using five fuzzy sets

Figure 6.5 Fitness graph for Seller agents in FPSB-PV auctions

Figure 6.6 A Seller agent script written in SAML

Figure 6.7 SGML-based Seller agent scripts

Figure 6.8 State transition diagram for the Seller agent

Figure 7.1 The Bidder agent

Figure 7.2 Fitness graph for Bidder agents in English-CV auction

Figure 7.3 A Bidder agent script written in BAML
Figure 7.4 State transition diagram for the Bidder agent
Figure 7.5 Bidder agent exploring new auction sites in the IAS
Figure 8.1 Time taken by a Bidder agent to visit all running auction sites
Figure 8.2 Time taken by a Bidder agent to visit all running/non-running auction sites
Figure 8.3 Time taken by all Bidder agents to visit all three auction sites
Figure 8.4 Duration of auctions with different numbers of Bidder agents
Figure 8.5 Assessing the auctions played parameter
Figure 8.6 Assessing the population size and number of generations parameters
Figure 8.7 Assessing the gene allele parameter
Figure 8.8 Bidder agents' fitness for the auction methods and types
Figure 8.9 Bidding strategies for auction methods and types
Figure 8.10 Bidding strategies for auctions with various numbers of bidder opponents
Figure 8.11 Seller agents' fitness for auction methods and types
Figure 8.12 Selling strategies for auction methods and types
Figure A.1 Fuzzy set partitioning

Table 1.1 Auction methods
Table 1.2 Auction types
Table 2.1 Software agent properties
Table 3.1 Information required by a user to log into the agent system
Table 3.2 Message protocol in mobile agent system
Table 4.1 Agent environment simulator and Fuzzy-Genetic Algorithm parameters
Table 5.1 Auction simulator and Fuzzy-Genetic Algorithm parameters
Table 6.1 Seller agent's strategic decisions
Table 6.2 Fitness equations for Seller agents in simulated auctions
Table 6.3 Acceptable tag values for Seller agent scripts
Table 7.1 Bidder agent's strategic decisions
Table 7.2 Fitness equations for Bidder agents in simulated auctions
Table 7.3 Acceptable tag values for Bidder agent scripts
Table 8.1 Time taken by a Bidder agent to visit all known/unknown auction sites
Table 8.2 Duration of auctions using different auction methods
Table 8.3 Auction scenarios and their agent populations
Table 8.4 Assessing the crossover probability and mutation probability parameters
Table 8.5 Assessing the fuzzy partitioning parameters

Contents

Chapter 1 – Introduction

1.1 Thesis Overview 10

1.2 Introduction to Auctions..... 11

1.3 An Internet Auction Application..... 13

1.3.1 The Adaptable Mobile Agent.....14

1.3.2 The Internet Auction System.....16

1.3.3 Evaluation and Achievements 18

1.4 Guide to the rest of the Thesis..... 19

Chapter 2 – Survey of Existing Research

2.1 Software Agents..... 25

2.1.1 Mobile Agents.....27

2.1.2 Intelligent Agents.....30

2.2 Electronic Auctions and Multi-Agent Markets.....33

2.2.1 Semi-automated Online Auctions.....34

2.2.2 Automated Online Auctions.....34

2.2.3 Multi-Agent Markets.....36

2.2.4 Multi-Agent Auctions.....38

2.2.5 Electronic Negotiation.....39

2.3 Summary and Conclusion.....41

Chapter 3 – Creating and Managing Mobile Agents

3.1 Mobile Agent Infrastructure..... 43

3.1.1 Requirements.....43

3.1.2 Design Solutions 47

3.2 Managing Agents.....58

3.2.1 Requirements.....58

3.2.2 Design Solutions.....60

3.3 Achievements.....63

Chapter 4 – Creating Autonomous and Adaptable Agents

4.1 Automating Agents..... 65
4.1.1 Requirements..... 65
4.1.2 Design Solutions..... 67
4.2 Adapting Agents..... 74
4.2.1 Requirements..... 74
4.2.2 Design Solutions..... 76
4.3 Achievements..... 84

Chapter 5 - The Internet Auction System

5.1 Infrastructure..... 86
5.1.1 Requirements..... 86
5.1.2 Design Components..... 89
5.2 The Auction Simulator..... 101
5.2.1 Requirements..... 101
5.2.2 Design..... 103
5.3 Achievements..... 106

Chapter 6 – The Seller Agent

6.1 Overview..... 107
6.2 Utilisation and Implementation..... 108
6.2.1 Adaptation..... 108
6.2.2 Configuration..... 119
6.2.3 Automation..... 124
6.3 Agent Utilities..... 127

Chapter 7 – The Bidder Agent

7.1 Overview..... 128
7.2 Utilisation and Implementation..... 129
7.2.1 Adaptation..... 129
7.2.2 Configuration..... 135
7.2.3 Automation..... 141

Chapter 8 - Assessment

8.1 Evaluating the Internet Auction System..... 147

8.1.1 Testing for Bidder Agent Navigation.....148

8.1.2 Testing for Auction Duration.....152

8.2 Evaluating the Auction Simulator 154

8.2.1 Assumptions.....154

8.2.2 Investigating the Fuzzy-Genetic Algorithm Parameters..... 155

8.2.3 Evolving Bidding and Selling Strategies.....162

Chapter 9 - Conclusion

9.1 Thesis Overview..... 171

9.2 Evaluation..... 173

9.3 Further Work.....178

Appendix A - Fuzzy Set Partitioning..... 180

Appendix B - Derivation of Stationary Population Equation.....182

Appendix C - Source Code for the Auctioneer Monitor..... 183

Appendix D – Source Code for the Agent Router.....186

Appendix E – Partial Source Code for the Agent Client/Server..... 191

Appendix F – Partial Source Code for SAML and BAML Tag Checkers..... 197

Appendix G – Partial Source Code for Bidder and Seller Agent Interpreters.. 203

Appendix H – Partial Source Code for the Auction Simulator..... 215

References.....224

Chapter 1 - Introduction

In this chapter, the first section gives a brief overview of the thesis showing how auction theory and software agents are applied to an Internet auction application. The second section introduces auctions by explaining some of the different methods and types used to model real and simulated auctions. The third section describes how software agents are used to design and implement an Internet Auction System (IAS) that automates auctions over the Internet. The fourth section gives a chapter by chapter guide to the thesis, summarising the thesis objectives, research done, and achievements.

1.1 Thesis Overview

This thesis aims to automate various auction methods and types using the Internet and software agents. Initially, some background is given to these auction methods and types, mobile and intelligent software agents as well as electronic auctions. Then mobile and adaptable software agents are developed. These Adaptable Mobile Agents (AMA) are described in terms of tools and concepts. They are then applied to Internet auctions. In particular, AMA(s) are used in the design, implementation, and evaluation of an Internet Auction System (IAS). Figure 1.1 shows how auction theory, agent tools, and agent concepts are related to each other in this thesis.

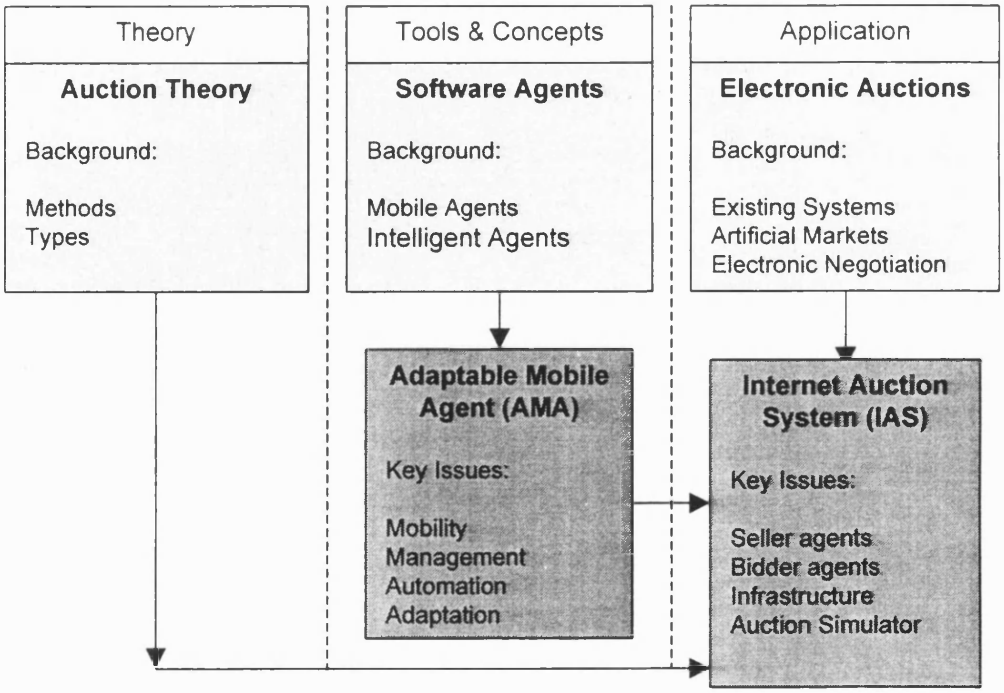


Figure 1.1 Thesis areas and overview

1.2 Introduction to Auctions

Auctions are a popular way to buy and sell a range of goods and services throughout the world. There are many types of auction and a variety of different ways to categorise and analyse them. For a good general description of auctions, see [CAS67]. For good introductions on auction theory, see [RAS96] and [BIN92]. Also, for some seminal papers on auction theory, see [VIC61], [MIL82], and [MCA87]. However, in this thesis, auctions are categorised in two ways. The first way categorises auctions by their bidding and selling rules, called the auction method (see the following table 1.1). In all auction methods, sellers may reserve the right not to sell their items below a certain amount. This is called a *reserve price*. If the highest bid is below this reserve price, the item is not sold.

Auction Method	Description of Bidding/Selling Rules
Sealed-bid	In a sealed-bid auction, bidders submit their sealed bids to an auctioneer. However, there are two further sub-types of sealed-bid auction. In a *first-price* sealed-bid (**FPSB**) auction, the highest bidder wins the auctioned item and pays his bid price. In a *second-price* sealed-bid auction, bidders submit their sealed bids to an auctioneer. The highest bidder wins the auctioned item but pays the second highest bid price. Another name for the second-price sealed-bid auction is the **Vickrey** auction. It is rarely used in practice but is interesting to model because unlike the other methods the second highest bid price is paid.
English	The English auction is often referred to as a *first-price open-cry* auction. In this auction, the auctioneer starts the bidding process by shouting out an asking price (called the *starting price*). If bidders accept to buy the item for the asking price, they may raise their hands (or signal to the auctioneer). Typically, only one bidder at a time signals. If the auctioneer receives a signal, he will then raise the asking price. This process repeats until the auctioneer does not receive any signals. The item is sold to the highest bidder at his bid price.
Dutch	The Dutch auction is often referred to as the *descending* auction. This auction is similar to the English auction in that the auctioneer shouts out an asking price. However, the auctioneer starts the bidding at a relatively high price for the item (*starting price*) and waits for any bidder to accept by signalling. If no bidder accepts, the auctioneer reduces the asking price. The process repeats until the first bidder signals to buy the item at the current asking price.

Table 1.1 Auction methods

The second way categorises auctions according to how bidders value the auctioned item, called the auction type (see the following table 1.2). In every auction, the auctioned item will have a

certain worth or *value* to each bidder. The item may not necessarily be worth the same to each bidder. In some auctions, bidders may be able to value the item precisely, e.g. antiques that will not be resold. In other auctions, bidders may not know the item's exact value, e.g. treasury bills. They must estimate its value. Their estimates are called their personal *valuations.*

Auction Type	Description of Bidders' Valuations of Auctioned Item
Private-value (PV)	In a *Private-value* (PV) auction, all the bidders know exactly what the item is worth to them, i.e. each bidder's value and valuation is the same. However, bidders may value the item differently, i.e. their *private values* may be different. This affects their bidding strategies.
Common-value (CV)	In a *Common-value* (CV) auction, the auctioned item has identical value to all bidders, but they may not know its true value. Therefore, they have to form their estimates (*valuations*) using their private information. To avoid over-bidding, and possibly inducing a loss (called *winner's curse*), they may '*scale-down*' their bids, see Sections 5.2.1 and 7.2.1.

Table 1.2 Auction types

Though the auction methods and types are categorised in two separate tables, auctions are specified by both their method and type. Therefore, with four auction methods and two auction types, eight auction method/type combinations are possible, e.g. Dutch-CV auction, Dutch-PV auction, English-CV auction etc. The eight auction method/types represent an interesting selection of auctions[1]. In particular, FPSB, English, and Dutch auction methods are chosen for automation for three reasons. Firstly, they are all used in the real world. Secondly, they represent the most popular auction methods used today. Some examples are: Estate Agencies selling houses in FPSB auctions; Sotheby's selling antiques in English auctions and fisherman selling their catch in Dutch auctions. Thirdly, they all have fixed simple bidding and selling rules. PV and CV auction types were chosen for automation because they are simpler to model. They are simplifications of real auctions because they make certain assumptions about how the bidders value the auctioned items. In PV auctions, bidders know exactly what the item is worth to them. They want the item for their own consumption and would not resell it, e.g. a desirable painting. Conversely, CV auctions are auctions in which bidders would resell the item for a profit, e.g. bulk tobacco. Since all the auction methods are fixed and the auction types simplify the modelling, it is reasonable to assume that they can be automated using computer technology. With the growing importance of electronic commerce, the business potential in fully automating electronic auctions is clear.

[1] In this thesis, automated auctions assume that bidders and sellers are risk neutral and only one item is auctioned at a time.

1.3 An Internet Auction Application

There are many varieties of electronic auctions. However, this thesis is concerned with the Online Web-based variety [AUC]. Within this variety, there are semi-automated and automated types [AGO96]. Semi-automated auctions enable bidders to submit their bids using web forms. However, they give users limited ability to control or configure their own auctions, e.g. Onsale [ONS]. On the other hand, automated auctions enable users to participate, configure and control their own auctions using web forms, e.g. Webalog [GOR97], Arizona WWWeb Works [ARZ] and AuctionBot [WU98a], [WU98b], [WEL98]. Most Online Web-based auctions use the English method; a few use the Dutch method, e.g. Klik-Klok [KLK]. In any case, none of these commercial systems is agent-based. It appears that agent-based auction systems have yet to be commercialised.

However, a few research projects are investigating ways to automate electronic commerce systems using agents. In some projects, researchers are building artificial markets [ERI97], [COLI97] where agents negotiate to buy and sell goods, e.g. Kasbah [CHA96] and Fishmarket Auction [FM98], [ROD98]. In other projects, researchers are focusing on the theoretical foundations by formalising electronic negotiation between agents [BEA96a], [BEA96b]. In particular, some research is investigating Game Theory as a basis for deriving agent strategies. Others are investigating Artificial Intelligence techniques, like Genetic Algorithms (GA) to derive an agent's strategies [DW95], [DW96a], [DW96b], [KIM96], [OLI97a], and [OLI97b].

Reviewing existing Web-based auction systems and techniques highlighted three points. Firstly, the lack of agent-based electronic auction systems; in particular none were constructed using mobile agents and this may be an interesting (and different) way to construct them. Secondly, the inflexibility of existing electronic auctions. Thirdly, the inherent complexity associated with distributed multi-agent systems [WOO98a], [WOO98b], [BIN97], [FIS93], and [ADE]. These points provided the motivation to design a novel Web-based auction system based on novel Adaptable Mobile Agents (AMA).

The next section overviews the issues raised by designing the AMA. Though, the AMA will primarily be used to automate auctions, they are described in terms of tools and concepts. In particular, agent mobility, management, automation and adaptation are covered. The intention is to design flexible AMA(s) that could be used to automate other types of electronic commerce system besides electronic auctions (e.g. electronic stock markets).

1.3.1 The Adaptable Mobile Agent

There is a vast array of literature on software agents. There are many definitions of what an agent is and many ways to classify their properties and features. For details on agent definitions and classifications, see [JEN98a], [JEN98b], [NWA96], and [WOO96]. Equally, there is a vast array of literature on agent applications, see [IAL1] and [IAL2]. Therefore, only issues important to designing AMA(s) for electronic auction applications are investigated (see Chapter 2 for background on mobile/intelligent agents). The four main agent issues covered in this thesis are:

- Mobility;
- Management;
- Automation;
- Adaptation.

Agent mobility concerns how agents move from computer to computer over networks, e.g. Internet. Software agents can be written in general purpose languages (e.g. Java or Perl) [BOS97] or languages specific to mobile agents (e.g. Telescript) or specific to multi-agent systems (e.g. KQML). However, AMA(s) are implemented as text-based (interpretable) scripts written in specially designed Standardised Generalised Mark-up Language (SGML) languages[2] [SMI92]. Text-based scripts could make the task of defining an agent's properties easier by storing its properties in tags that are understandable. Moreover, scripting agents using SGML languages provides a method of keeping agents consistent, verifiable and standardised. To implement mobile agents, an agent infrastructure is required. The AMA(s) infrastructure consists of web servers [SPA96], Common Gateway Interfaces (CGI) scripts [CGI], sockets, [GAR94], and Java-based clients/servers [COH96]. It enables AMA(s) to move freely between computers. Central to this infrastructure is the Hypertext Transfer Protocol (HTTP). In particular, the HTTP 'Post' request is used to transport AMA(s) around the network [LIN96]. This infrastructure has its advantages. It is platform independent, dynamic, supports mobile users, enables HTTP security features to be adopted, and uses web servers as a backbone for the mobile agent network.

Agent management concerns how mobile agents are created monitored and controlled. Most user-agent management tasks can be achieved by using browser-type Graphical User Interfaces (GUI), e.g. Netscape browser [KRE96]. Users can create, run, and monitor agents using the standard GUI(s). As AMA(s) move from computer to computer, they send messages back to their users, informing them of their status, location etc. All agent transport and agent to user communications can be handled by HTTP requests.

---

2 In this thesis, only script languages for Bidder and Seller agents are designed.

Agent automation concerns how agents run on computers and perform tasks. AMA(s) are modelled as finite automata. They begin in a start state and finish in a terminating state with various state transitions between. Events within a system cause AMA(s) to change state. As they do so, they perform actions (e.g. update its user with its location). AMA(s) are implemented as automata by embedding their states within their scripts, allowing them to be self-modifiable so that they can change their own state and by using agent interpreters to perform their actions. AMA(s) can only communicate with other AMA(s) if they are on the same computer. Computer programs called agent monitors[3] control and mediate all agent communications. Every computer in the network would have at least one agent monitor to mediate its resident agents. This enforces agent synchronisation, avoids the complexity inherent with distributed agent systems, and reduces network traffic. Only messages sent by the agent back to its user are sent between computers. Even this can be reduced because agents update their own scripts with information (logging what they have done) and take it back to their users to read.

Agent adaptation[4] concerns how users adapt their agents to do their tasks better. If users are going to trust AMA(s) to perform tasks on their behalf, they require some way of testing and adapting them. The method used in this thesis is to design AMA(s) that embed fuzzy-genetic task-based strategies [BI96], [HUR97] within their scripts. These strategies could then be evolved in a purpose built simulator using Fuzzy-Genetic Algorithms (FGA) [ZAD74], [KOK94], [HOL92], [GOL89]. This would enable users to test and evolve their AMA(s) before they are allowed to perform tasks in the real agent-based system.

These issues are important to electronic commerce systems because these systems tend to be distributed and 'information rich' [KAL97]. Therefore, mobile agents could be used to engineer these distributed systems. If they are manageable, they could be used to monitor and control information. If they automate tasks, they could be used to perform their users' tasks anywhere within the distributed system. Finally, if they are adaptable, users could test and improve their agents before releasing them into the distributed system. Moreover, agents with all these properties (i.e. AMA(s)) could be used to automate electronic commerce systems.

Designing AMA(s) made three specific achievements. Firstly, the AMA infrastructure successfully implemented platform independence, agent mobility, user mobility, and rudimentary security. Secondly, the idea that SGML could be used to mark-up an agent's features strategies and states was simple and effective. Thirdly, the idea that users adapt their agents' strategies in

---

3 Agent monitors are programs that control multithreaded agent systems by synchronising the concurrently running agent threads.

4 In this thesis, an 'adaptable' agent is an agent that can have its strategies adapted by a user in some kind of simulator. Whilst an 'adaptive' agent is an agent that can adapt its strategies itself.

agent environment simulators using FGA(s) before releasing them into the real agent system was both flexible and effective. Moreover, the AMA tools and concepts discussed in this section are used as a basis for automating auctions over the Internet. AMA(s) are used to design and build a working Internet Auction System (IAS). This is discussed in the next section.

1.3.2 The Internet Auction System

The IAS is a distributed, multithreaded, client-server system[5] that uses software agents. It uses a range of technologies that form an effective and novel electronic auction system. It enables users to automate concurrent real-time auctions on their computers using Seller agents and participate in remote auctions using Bidder agents, see figure 1.2.

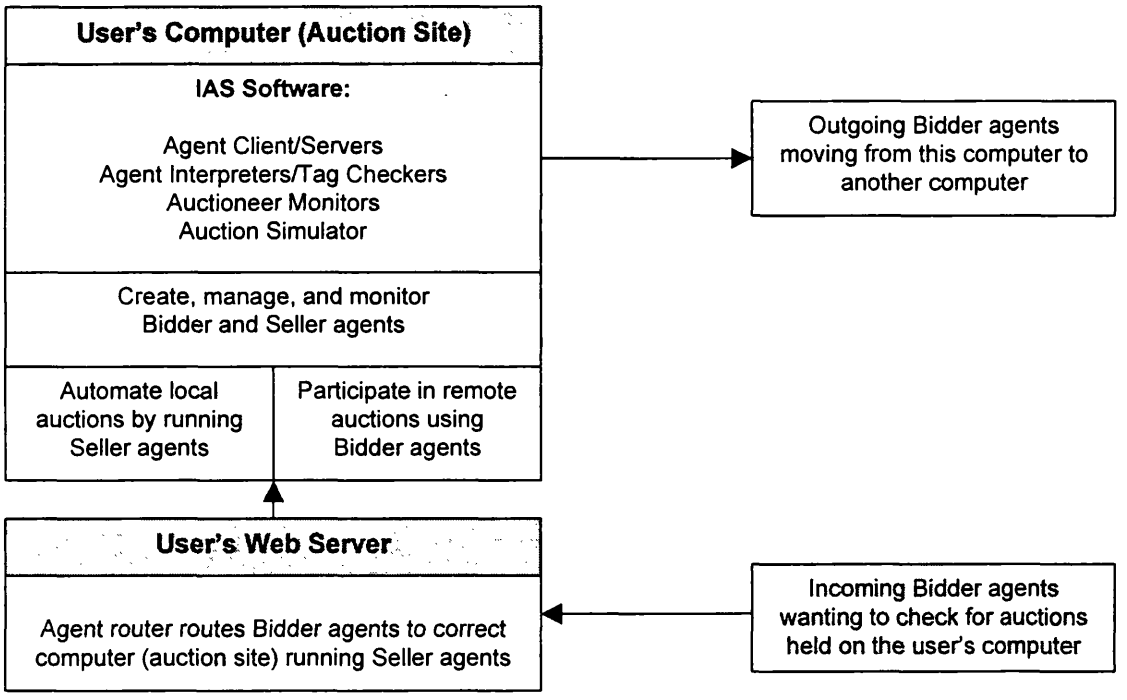


Figure 1.2 A user's computer connected the IAS to form an auction site[6]

Seller agents are implemented as adaptable agents; they are not mobile. A user automates the selling part of an auction by configuring his Seller agents and running them locally on his computer. This is called a user's *auction site*. Users may determine various selling criteria:

- Item to be auctioned and its estimated or actual value;
- Auction method;

5 The IAS consists of approximately eight thousand lines of Java source code, see appendices.
6 Each web server may or may not be on the same computer as the user's auction site. The web server may route Bidder agents to any number of auction sites held on one or more computers.

- The date and time that the auction shall start;
- The selling strategies that the Seller agents shall use.

A user sets up an auction in two stages. During the first stage, a user evolves satisfactory selling strategies in an auction simulator. During the second stage, a user configures a Seller agent with the evolved strategies and other auction parameters using browser-like GUI(s). Once a user has created his Seller agent[7], he may automate the selling part of an auction by running the Seller agent. A user can monitor his Seller agent before, during, and after the auction. Once a Seller agent has completed its auction, it stops running.

Bidder agents are implemented as AMA(s); they are mobile. A user automates the process of searching, exploring, and bidding in remote auctions by configuring his Bidder agents and running them. Users may determine various bidding criteria:

- Item to bid for at an auction;
- Schedule or list of auction sites (addresses)[8] to try out;
- The date and time that the chosen Seller agent's auction should start by;
- The bidding strategies that the Bidder agents shall use?

Creating Bidder agents is similar to creating Seller agents. Initially, a user evolves satisfactory bidding strategies in an auction simulator and then configures the Bidder agent with its other auction parameters. Once a user has created his Bidder agent, he may automate the searching, exploring and bidding part of the auction system by running the Bidder agent. A user can monitor his Bidder agent wherever it is and whatever it is doing from browser-like GUI(s). A user may create and simultaneously run any number of Bidder and Seller agents. As more users create Bidder and Seller agents, they form a dynamic networked IAS.

Each user within the IAS requires various software components to manage and run agents. The five main components are the agent client/server, agent router, agent interpreter, auctioneer monitor, and auction simulator. The agent client/server and its GUI(s) enable users to manage and monitor their agents. Together, agent client/servers and routers form the mobile agent infrastructure. Bidder agents are free to move within this infrastructure seeking remote and distributed auctions. Both Seller and Bidder agent scripts are interpreted. Seller agent scripts run

---

7 In real auctions, speed is often an important issue. Since most Web-based auctions take days to complete, it is intended that Seller agents' auctions would take minutes. However, unlike some real auctions, only one item is auctioned at a time and the item's type is ignored.

8 Each user within the IAS is capable of running a separate auction site with a unique address. Users may run one or more concurrent auctions by creating and running Seller agents.

locally on a user's computer. Bidder agent scripts are mobile but still run locally on the computers they happen to be visiting. Auctioneer monitors are used to conduct auctions between resident Seller agents and visiting Bidder agents by mediating and synchronising all Bidder-Seller agent communications, i.e. bidding and auctioning. The auction simulator and its FGA enable users to evolve bidding and selling strategies[9] in various auction environments by setting a range of parameters and assumptions. Their agents can then use these strategies in real IAS auctions.

1.3.3 Evaluation and Achievements

The IAS was evaluated using test sets. The first test sets investigated the time taken by Bidder agents to navigate through the IAS and the time taken to automate the auctions. The second test sets investigated what effect the various auction simulator and FGA parameters had on an evolving agent population's average fitness. The third test sets investigated bidding and selling strategies evolved in various auction environments.

The results of the first test sets demonstrated that mobile Bidder agents could successfully navigate through an IAS visiting auction sites. Bidder agents managed to cope with incorrect auction sites and complex schedules. In all the tests, ranging from an IAS with many auction sites to an IAS with many concurrently running Bidder agents, the system was efficient and stable. In most cases, the Bidder agents completed their schedules in a matter of minutes. Similarly, the automated auctions were completed in most cases in less than a minute. The IAS successfully automated auctions over the Internet using agents in an efficient and effective way.

The results of the second test sets demonstrated that the main auction parameters (number of generations, population size, auctions played, mutation probability and crossover probability) had an important affect on an evolving population's fitness. Most of the results were expected. They showed that increasing population size, increasing the number of generations, etc., generally increased a population's fitness. However, crossover and mutation probability caused more subtle changes to a population's fitness. Next, parameter values were specifically chosen to evolve bidding and selling strategies in various auction environments. The two main reasons for choosing preferred values were, firstly, to improve the FGA so that it evolved fitter agents and, secondly, to make the task of analysing the evolved strategies easier.

---

[9]Specifically, bidding strategies determine the amounts the Bidder agent bids, scale-downs, and selling strategies determine the amounts the Seller agent sets for its starting and reserve prices.

The results of the third test sets demonstrated that the auction simulator and FGA could evolve bidding and selling strategies that were profitable (and in most cases rational). However, the FGA was more successful for PV auctions than for CV auctions. The strategies evolved for CV auctions appeared complex and drawing any firm conclusions to their success was difficult. However, in nearly all the simulated auction environments, the agents made a profit. In this respect, the auction simulator and the FGA were successful. Further analysis was required to assess the accuracy and relevancy of the CV strategies.

Designing, building and testing the IAS highlighted three specific achievements. Firstly, SGML-based AMA(s) were used to form novel Bidder and Seller agents. This provided users with an easy way to script their Bidder and Seller agents. Secondly, all auction methods and types were fully automated using Bidder and Seller agents, multithreaded auctioneer monitors, and agent interpreters. Consequently, Bidder and Seller agents were synchronised, inter-agent communications between computers was eliminated, and network traffic was reduced. This provided a mechanism to automate auctions very quickly, i.e. minutes. Thirdly, an auction simulator and its FGA were partially successfully. They evolved rational bidding and selling strategies for PV auctions but evolved 'inconclusive' strategies for CV auctions.

However, there is plenty of scope for further research on agent-based electronic auction/commerce systems. For example, Bidder and Seller strategies could be made more sophisticated. The IAS could be made more interactive and AMA(s) could be used to design and build other electronic commerce systems.

1.4 Guide to the rest of the Thesis

This section guides the reader through the rest of the thesis and highlights each chapter's contribution to the thesis objectives, the research done, and its achievements. The thesis falls into four parts: *Background, Designing Adaptable Mobile Agents, An Internet Auction Application,* and *Analysis.* The first part (Chapters 1 and 2) explores areas important to the thesis. The second part (Chapters 3 and 4) concentrates on developing the AMA. The third part (Chapters 5, 6, and 7) concentrates on applying the tools and concepts from the second part to building an IAS. The fourth part (Chapters 8 and 9) evaluates and concludes the thesis.

Chapter 2 – Survey of Existing Research

Objectives: The aim of this chapter was to investigate some current agent technologies and electronic auctions. This helped to identify good research areas and focus the thesis.

Research: Agents were classified in many ways: by what applications they were designed for, by their behaviour, or by the techniques they employed. AMA(s) were distinguished from existing agents by the combination of techniques they used, e.g. SGML, FGA(s), and multithreaded agent monitors. Most electronic auctions reviewed were Web-based. However, very few auction systems were agent-based, Fishmarket [ROD98] being the exception. Although, in all the electronic auctions researched none used AMA(s) to automate their auctions.

Achievements: It appeared mobile agent-based electronic auctions were an unexplored area. Therefore, this thesis aimed to design a novel agent-based electronic auction system using AMA(s). However, first the AMA(s) tools and concepts needed to be developed.

Chapter 3 – Creating and Managing Mobile Agents

Objectives: The aim of this chapter and the following chapter was to design the AMA in terms of tools and concepts. In this chapter, a mobile agent infrastructure using the Web and Internet was proposed. This infrastructure should enable users to manage and monitor their mobile agents as they move from computer to computer.

Research: The mobile agent infrastructure used web servers/CGI as agent routers, sockets, and agent client/servers to form a mobile agent infrastructure. Agents used HTTP requests to transport themselves around the agent network and communicate with their users. Designing agents to be text-based scripts (as opposed to objects) made the agents easier to transport.

Achievements: A Web-based infrastructure for mobile agents was successfully created. Users could join and leave dynamic and transient agent networks. In addition, they could create mobile agents and monitor them within the distributed agent system.

Chapter 4 – Creating Autonomous and Adaptable Agents

Objectives: This chapter completed the development of the AMA. Its aim was to design agents that would automate strategic-based tasks (decisions). Moreover, the agents should be easy to script and possess adaptable strategies that could be tested in an agent environment simulator.

Research: AMA(s) were implemented as text-based scripts and modelled as finite automata. Moreover, an agent designed for a particular application would have all its features marked up using a tailor-made SGML mark-up language (see sections 6.2 & 7.2 for SGML-based languages for Bidder and Seller agents). These scripted agents were self-modifiable and encoded their

states. Agent interpreters were used to run the scripts and change their states as they automated processes. To make an agent adaptable, it required strategies that could be evaluated and evolved. Simulators could be used to evaluate an agent's strategies and FGA(s) could be used to evolve them. Since a simulator depends on the agent's application area, only concepts were investigated.

Achievements: At this point in the thesis, it was shown (conceptually) that agents could be made mobile, manageable, autonomous, and adaptable. These agent tools and concepts were then used to design the IAS.

Chapter 5 - The Internet Auction System

Objectives: The aim of this and the following two chapters was to use AMA(s) to automate auctions and build the IAS. (This chapter describes the IAS architecture whilst Chapters 6 and 7 describe the agents in detail). Various problems specific to an auction system were tackled, e.g. how are auction sites uniquely addressed? How do Bidder agents explore and register? How are multithreaded auctions conducted? How are auctions simulated within an auction simulator?

Research: The IAS was designed so that it overcame the technical problems described above. GUI(s) were used to create and manage agents. Agent clients/servers and routers were used to build the infrastructure for mobile agents. Agent interpreters and auctioneer monitors were used to automate agents and conduct the auctions. Auction simulators and FGA(s) were used to adapt bidding and selling strategies in various simulated auction environments.

Achievements: The IAS successfully used all the tools and concepts developed earlier in Chapters 3 and 4. As, with most systems, improvements could be made. For example, the interfaces used to create and monitor the auctions could be animated. However, the aim of this thesis was to fully automate auctions using AMA(s) and this was achieved.

Chapter 6 – The Seller Agent

Objectives: This aim of this chapter was to specify the Seller agent in terms of its configurable parameters and components. Users should be able to determine when the auction starts using their chosen auction method. In addition, users should be able to adapt selling strategies like starting and reserve prices for their Seller agents. Though the Seller agent was not mobile, its life cycle from creation to post auction still required formalising in terms of states, state transitions, events, and actions.

Research: The Seller agent was modelled as a finite automaton and constructed using a specifically designed SGML script language. This simple mark-up language, called SAML (Seller Mark-up Language), was used to write Seller agent scripts. Information such as identification, auction assumptions, fuzzy-genetic strategies, and memory were marked up. As the Seller agent could modify its own script, it could log various details such as Bidder agents registering to join its auction, actual bids offered and the auction outcome.

Achievements: The Seller agent was successfully implemented. A simple mark-up language, SAML was used to script Seller agents. Together with its interpreter, the Seller agent initiated auctions, made decisions, and kept track of events. Designing the Seller agent as a self-modifying (automaton) script made it a powerful tool in automating the seller side of an auction.

Chapter 7 – The Bidder Agent

Objectives: The aim of this chapter was to specify Bidder agents in terms of its configurable parameters and components. Mobile Bidder agents were required to locate remote auctions and make decisions on which auctions to bid in. Some problems specific to Bidder agents were how do Bidder agents find new auctions. How do they decide which auctions to go to? These and other issues were formalised in terms of states, state transitions, events, and actions.

Research: Similarly to the Seller agent, the Bidder agent was modelled as a finite automaton and constructed using a specifically designed SGML script language. This simple mark-up language, called BAML (Bidder Mark-up Language), was used to write Bidder agent scripts. However, Bidder agents were required to keep track of where they went and what auctions they found. In the IAS, Bidder agents co-operated by sharing their auction site addresses with each other. This helped them to explore new auctions. As Bidder agents moved around the IAS, they kept track of all auctions they visited and any bidding they may have done within them.

Achievements: The Bidder agent was successfully implemented. A simple mark-up language, BAML was used to script Bidder agents. Together with its interpreter, the Bidder searched for remote auctions, made decisions on which auctions to register with and bid in. Designing the Bidder agent as a self-modifying (automaton) script made it a powerful tool in automating the buyer side of an auction.

Chapter 8 – Assessment

Objectives: The thesis was evaluated in three ways. Firstly, by how well the Bidder agents navigated various IAS(s). Secondly, by how well Bidder and Seller agents automated auctions. Thirdly, by how well the auction simulator and FGA evolved profitable and rational bidding and selling strategies.

Research: The IAS was tested in two ways. Firstly, to see how long it took Bidder agents to navigate through an IAS with various numbers of concurrently running Bidder and Seller agents. Secondly, to see how long it took to automate auctions of various methods with various numbers of Bidder agents. The auction simulator and FGA were also tested in two ways. Firstly, various FGA and auction parameters were tested to see their affect on the fitness (profitability) of evolving agent populations. Secondly, suitably selected parameters were used to evolve bidding and selling strategies in specific auction environments.

Achievements: The IAS was successful in three respects. Firstly, it enabled users to use Bidder agents as an effective tool in searching for remote (Seller agents) auctions. The times taken by the Bidder agents were measured in minutes. Secondly, it automated all the auction methods with various numbers of Bidder agents in an acceptable time. Most auctions took minutes. As expected, as agent populations rose, the IAS slowed. However, in theory it could manage unlimited populations of Bidder and Seller agents. Thirdly, the auction simulator successfully evolved profitable (and mostly) rational bidding and selling strategies for all the auction methods. However, though the evolved strategies for PV auctions were verified by auction theory, the evolved strategies for CV auctions were inconclusive.

Chapter 9 – Conclusion

Objectives: This thesis had three specific aims. Firstly, to design novel AMA(s). Secondly, to use AMA(s) to automate auctions within an IAS. Thirdly, to build an FGA-based auction simulator, enabling users to evolve bidding and selling strategies for their agents.

Research: The AMA(s) were designed to be mobile, manageable, autonomous, and adaptable. An infrastructure for AMA(s) was built using the Internet, web servers and agent client/servers and routers. Agents were scripted using SGML-based mark-up languages, run using agent interpreters, controlled using agent monitors and adapted using FGA-based simulators. These tools and concepts were used to build the IAS. The IAS was then evaluated by testing how effectively Bidder agents navigated various IAS(s) and how long the auctions took with different

populations of Seller and Bidder agents. Lastly, the auction simulator was tested to see if it evolved rational and profitable bidding and selling strategies.

Achievements: Firstly, novel AMA(s) were designed using various techniques, e.g. SGML, finite automata, and fuzzy-genetic adaptable strategies. Secondly, the AMA(s) were used to design and implement an IAS. This system successfully automated auctions efficiently. Thirdly, the auction simulator evolved profitable (and mostly rational) bidding and selling strategies.

Chapter 2 – Survey of Existing Research

This chapter reviews some research on software agents and electronic auctions. It aims to categorise agents, compare their methodologies, and expose potentially novel areas for agent-based systems. Furthermore, it provides a basis for designing Adaptable Mobile Agents (AMA) and implementing them in an Internet Auction System (IAS). Software agents are examined first. In particular, mobile and intelligent agents are investigated. Next online electronic auctions are examined with particular emphasis on investigating the automated agent-based variety.

2.1 Software agents

Software agents are difficult to define because they are ubiquitous. They have been applied to a variety of applications. Consequently, they may possess a variety of properties. In the research literature, many definitions and applications can be found. The references [IAL1] and [IAL2] list research on intelligent agents. Within these references, some authors categorise intelligent agents according to their properties. Others focus on agent applications as a basis for categorisation [FRA]. These can be further sub-classified into what an agent's control structures, intended environments, languages, and applications are. For example, Nwana categorises agents into the following types [NWA96]:

- Collaborative;
- Interface;
- Mobile;
- Information;
- Reactive;
- Heterogeneous;
- Smart.

Other agent researchers like Gilbert acknowledge the problems with categorising agents. Therefore, Gilbert describes agents using a simplified three dimensional model [GIL], [GIL97]:

- Agency (an agent's level of interaction);
- Mobility (an agent's ability to move);
- Intelligence (an agent's ability to reason, plan and learn).

However, a good way to understand software agents is to list some of their properties and describe them in the context of an agent-based system like an Internet auction. The following table 2.1 lists agent properties according to Franklin [FRA] but uses an Internet auction example.

AGENT PROPERTY	DESCRIPTION (Example of agents in an electronic auction)
Autonomous	The agents conduct auctions independently of the users. In an electronic auction, agents would bid and sell automatically. Bidder agents would bid and Seller agents would sell within the auction without user intervention.
Goal-oriented	The agents have specific goals. In the auction example, Bidder and Seller agents want to maximise their profit at an auction.
Communicative	How agents communicate with their users and between themselves is complex. However, in an electronic auction, agents must communicate with each other during bidding and inform their users of what they are doing at the auction.
Adaptable (rather than adaptive)	Adaptive agents alter their behaviour in response to their changing environment. Whilst, adaptable agents can be modified by the user in some kind of artificial environment and then sent out into their real environment. In the case of Bidder and Seller agents, an auction simulator would be used. However, these agents do not learn whilst in their auction environment, they are not adaptive within the auction.
Mobile	Mobile agents can move from computer to computer over a computer network like the Internet. In the electronic auction, Bidder agents could move from computer to computer searching for Seller agent auctions.
Flexible	Designing flexible agents is very difficult [WOO98a]. The agents designed in this thesis are intended for electronic commerce systems. Broadening their potential application domain could be very difficult.
Multi-agent	Since agents within a distributed electronic auction interact (bidding and selling), they form a multi-agent system. The complication with multi-agent systems is trying to ensure the system behaves as expected. This is more difficult because agents run concurrently and the various ways agents interact can become large [FIS93], [WOO92].
Reactive	Reactive agents are responsive to their environment. In the electronic auction example, Bidder agents would be reactive to the changing auction system environment. One example of this could be that an auction has finished and the Bidder agent searches for another auction.

Table 2.1 Software agent properties

The lists of agent definitions, properties, and applications are as varied as the agents themselves. All the categorisations and classifications listed above appear to be a good way of understanding agents. However, in this thesis, agent properties are simplified, categorised and investigated according to the following:

- Mobility;
- Management;
- Automation;
- Adaptation.

These categories focus on agent properties and features that are particularly relevant to an AMA. They are the core features that an AMA would require in order to automate an electronic commerce system and they are explored in detail in Chapters 3 and 4. However, in much of the agent literature, agents are labelled as mobile and/or intelligent. The next two sections review existing mobile and intelligent agents in more detail and attempt to explain their methodologies and applications[1].

2.1.1 Mobile Agents

This section briefly looks at some mobile agent issues. It highlights the similarities and differences between current mobile agents and the AMA prototype. However, mobile agent research is a large and varied research area, therefore, this section focuses on three main issues. Firstly, the benefits and problems associated with using mobile agents. Secondly, how mobile agents are used to implement and model distributed systems. Thirdly, what the mobile agent applications are. However, security issues are for the most part ignored. This includes agent authentication and encryption.

The Benefits and Problems

Nwana lists a few benefits and problems associated with mobile agents [NWA96]. The main benefits are a reduction in inter-agent communications between computers; asynchronous computing and a flexible approach to distributed system design. Intra-agent communications are reduced because mobile agents do not need dedicated connections with each other or other computers. Mobile agents can communicate locally on the same computer using standard methods like pipes or using shared resources [WAL92]. However, access to these shared resources must be controlled. Agent monitor programs can be used to mediate and control all

[1] For a more rigorous review of agent theories and languages, see [WOO95], [WOO96], and [JEN98a].

messaging between agents on a local computer system. Writing programs to control multithreaded agent applications is difficult [COH96] but it is fair to say that co-ordinating a group of local agents is easier than co-ordinating a distributed group. Furthermore, having mobile agents running on remote computers frees up users' time – a form of asynchronous programming. Lastly, mobile agents could introduce some flexibility into distributed system design. There is some truth in this especially when trying to design a multi-agent system in which all the agents can communicate remotely. The problems associated with agent synchronisation and communications reinforce the view that mobile agent systems appear to be a simpler approach.

However, [BEN97], [CHO96] highlights some pitfalls associated with using mobile agents. Mostly, they concern agent security, e.g. securing an agent's data or preventing an agent becoming a virus as it moves from computer to computer. Therefore, agents should be authenticated before they reach another computer. They should be stable and not cause the local system to crash. They should have their access to resources and information restricted. Moreover, they should have information held within their own scripts protected from other agents. These problems are being tackled using scripting languages like Safe-TCL. However, while these problems are not tackled in this thesis, they would provide an interesting and important extension to this thesis.

Methodologies

In multi-agent systems, agents may communicate, co-operate, and compete. They may do this synchronously or asynchronously. Exactly how they do this depends on their application environment and the protocol used for inter-agent communication. In some systems, agents may collaborate and share information [NWA96]. In other systems, notably electronic markets, agents compete. However, there are two popular ways to construct agent-based distributed systems [CHO96]. The first way is to have mobile agents that move from computer to computer and interact within a system and its agents locally. The second way is to have static agents (at each computer) communicating to each other using some agent protocol.

The first way to construct agent-based distributed systems (procedural approach) requires a language for the agents to be written in. General purpose programming languages can be used such as C++, Perl, Java or Safe-TCL. In particular, Java applets have been used to design mobile applications. However, there are languages specifically designed for mobile agent applications. Some examples are Telescript, Aglets [LAN96], and CyberAgent [HAR95]. Telescript by

General Magic[2] is the most well known mobile agent scripting language. It is an interpreted, object-oriented language specifically used in the design of mobile agent applications. Briefly, Telescript enables users to model their systems using secure mobile agents and 'places'. Agents may meet at these 'places' to share resources and communicate. Every 'place' has the software (interpreter or engine) running, controlling and authenticating visiting agents. (However, Telescript did not take off, primarily, because the demand for mobile agent systems has not been sufficient). Another mobile agent language is Aglets by IBM. Aglets are very similar to Telescript agents but are based on Java applets. Aglets have useful Graphic User Interfaces (GUI) that provide users with an easy way to monitor and manage agents. It was the first visual agent based environment for network-based applications.

The second way to construct agent-based distributed systems (declarative approach) requires agents to communicate using an agent communication protocol. Agents send each other information (definitions, statements, or assumptions) over computer networks (subject to the agent protocol) and use this information to perform their tasks. A notable example is Agent Communication Language developed by ARPA. This consists of an agent vocabulary, a language called Knowledge Interchange Format (KIF) and a language called Knowledge Query and Manipulation Language (KQML) [CHO96], [CHA92]. KIF is based on first-order predicate calculus for encoding rules and expressions. Compared to KIF, KQML is less abstract. It provides agents with methods to communicate linguistically. Together, they form a protocol for all inter-agent communication. However, since these methods are used to design systems that use stationary agents, they will not be explored any further.

Implementing mobile agents is problematic. Firstly, how do agents move around from computer to computer? As described in the introduction, Lingnau [LIN96] suggested an HTTP-based infrastructure. This is central to creating an infrastructure or agent network for the AMA, see Section 3.1. Secondly, how do agents know where to go? Users could give their agents a list of computer/user addresses within the agent network, or the agents could explore the agent network or they could move around subject to their specific goals. The AMA system is designed to be dynamic and transient (users may join and leave the distributed system from their computers). AMA(s) are designed to have user-specified schedule of hosts. However, they can explore new computers or users if they co-operate and share their knowledge of computer addresses with other agents. In the case of the electronic auction, AMA(s) would have specific goals, i.e. go to the best auction subject to some criteria. This idea will be explored in Chapter 4 and implemented in Chapter 7.

[2] The company General Magic has since been liquidated.

Applications

Computer systems that use mobile agents are currently unpopular. There are very few commercial systems being used because of security reasons. However, Sony's Magic Link (Personal Digital Assistant) is used to assist a user with their fax, e-mail, and telephone communications [NWA96]. France Telecom is planning to use Telescript agents for seeking cheap railway tickets and car hiring [NWA96]. The mobile agents search the network subject to time and price constraints. In addition, BT Research Labs have successfully experimented with mobile agents to dynamically and adaptively route messages [BT]. However, it appears there are no electronic auction systems that use mobile agents. Although, there are agents (not mobile) being used in electronic commerce systems such as BarginFinder from Anderson Consulting, Kasbah marketplace by Maes at MIT [CHA96] and ShopBot from University of Washington [CHE96]. Therefore, an electronic auction system that automates auctions over the Internet using mobile agents would be a novel mobile agent-based system.

2.1.2 Intelligent Agents

This section briefly looks at intelligent agents (whether mobile or immobile). Firstly, it describes some of the characteristics associated with 'intelligent' agents. Next, it describes how the techniques used by an 'adaptable' agent for an electronic commerce system, compare with the current intelligent agent techniques. Finally, this section highlights a few applications where intelligent agents are being used. For an overview of intelligent agent applications, see [GIL], [GIL97], [WOO95], and [WOO96].

Characterisation

As with generic software agents, the 'intelligent' varieties are equally difficult to define. Partly, because intelligence is a concept debated by researchers in many disciplines inside and outside computer science and partly because intelligence is so loosely accredited to many rather mundane agents. Gilbert says that intelligent agents are all autonomous, goal-driven and reactive [GIL]. An agent is autonomous if it can do tasks independently of its user. A goal-driven agent as the name suggests has specific goals to achieve. These can be specified by the user in different ways. Some agents have their goals defined in scripts. Some agent programs run in such a way that their goals are reached through executing the program. Some agents have rules that they follow (many if-then statements or logical rules). In addition, some have embedded goals with some planning and adaptability. However, agents must react to their environment to exhibit

intelligent behaviour. How they react depends on their goals. Lastly, agents should continue to run whether a user is present or not.

Gilbert goes on to list a few sub-intelligent agent categories: social, adaptive, mobile, and believable. These may or may not be present in an intelligent agent system. Social agents communicate and co-operate with each other to perform collaborative tasks. They need to communicate with each other. Some agent communication languages were described in the last section, KQML being one of them. Mobile agents have already been discussed in the last section. Believable agents are forming a branch of agent technology that deals with an agent's personality and morphology [BIC98]. Finally, constructing adaptive agents requires investigating how agents learn to adapt to their environments. This is providing a renaissance for Artificial Intelligence because techniques like Neural Nets and Genetic Algorithms are being used, see [GOL89], [HOL92], [OLE97] and [PAT90].

For agents to learn there needs to be a way to assess their performance and improve it. For example, four components are necessary for assessing an agent's ability to do a task. These are a performance element, a learning element, a critical element, and a problem generator. The first component defines what agent tasks and actions are to be assessed. The second component defines how agents can improve their performance in doing their tasks. The third component defines the measurements used to assess how well they did. This should be consistent and fair to participating agents. The fourth component defines how agents discover new or better ways to do something. All these problems are providing fruitful research areas.

However, the agents intended for electronic commerce systems and in particular electronic auctions will be adaptable; i.e. users should be able to adapt their agents before using them. The agent uses embedded rules and goals in the form of fuzzy-genetic strategies to make decisions. They are not adaptive in the sense that they plan and learn whilst in their environment. Instead, they possess user-adaptable strategies that can be evolved and modified in an agent environment simulator. This is a good way for users to test their intelligent agents before releasing them into the real agent system. These ideas are described in Section 4.2.

Applications

Intelligent agent research and applications are on the increase. It appears that almost every university has some agent research [IAL2]. For example, Carnegie Mellon University is looking at believable agents (called the OZ project). MIT also has various research projects. Maes and Brooks are looking at the non-symbolic side of agents, i.e. behavioural side. Wooldridge and

Jennings are formalising agent theories and languages at the Distributed AI centre at Queen Mary and Westfield College, University of London [MKT]. The University of Maryland (UMBC) [UMB] has a range of agent research, e.g. KQML. In any case, intelligent agents are being used in applications areas as diverse as telecommunications to business process re-engineering to electronic commerce [GIL97]. Gilbert specifies eight distinct application areas. These are Systems and Network Management; Mobile Access; Mail and Messaging; Information Access; Collaboration; Workflow; Electronic Commerce and Adaptive User Interfaces. Each of these areas is being extensively researched with a few being marketed as commercial products.

In particular, the Internet and the World Wide Web (Web) have made electronic commerce possible, see [KAL96] for a good summary on intelligent agents for electronic commerce. For example, potentially, everyone with a PC connected to the Internet could participate in electronic markets. A user could become a virtual buyer. They could use agents to find and purchase goods or services from anyone that is connected to the Internet. Conversely, a user could become a virtual seller. They could use agents to advertise their products and services on their PC and wait for the buyers. Moreover, intelligent mobile agents could search for information, compare prices, negotiate for fair prices, and transact using credit card details. Internet malls, Internet auctions and virtual resellers are a few examples of where agent brokers could automate buying and selling processes. Agents could go shopping in the Internet mall. They could bid at Internet auctions and they could find the cheapest products from a Virtual reseller. Consumers and producers could form local or distributed markets that operate continuously. These electronic markets could be permanent or transient and agents could roam these markets without tiring. In fact, electronic auctions are already big business. One example is that American utilities will be able to auction their electricity directly to consumers. Large companies can then bid for electricity with bidding dependent on supply and demand. The whole process of bidding and billing will be handled by Power Exchange electronic auction [MAR97].

In addition, many of the large 'high-tech' corporations (Xerox Corp., IBM Corp., General Magic, AT&T and Zuno Ltd.[3]) are trying to commercialise intelligent agents. For example, Anderson Consulting has developed a few agent products called BarginFinder, LifeFinder, and InfoFinder. These search the Web for information and products specified by a user. Autonomy Corporation has developed agent technology called Agentware i3 that uses fuzzy logic and neural networks to 'cluster concepts' and 'dynamically reason'. There are many others [IAL1]. However, many companies for business reasons want to keep their ideas away from the public domain until they have fully developed their technology. Companies and customers need to be confident that their agents are going to work before using them. As mentioned in the last section besides the

3 Since writing this thesis, both General Magic and Zuno Ltd. have ceased to trade as companies.

technological difficulties associated with multi-agent systems, the biggest worry is security and payment authorisation. Whatever the drawbacks, multi-agent systems will play an important role in forming the next generation of electronic markets.

2.2 Electronic Auctions and Multi-Agent Markets

The last section introduced the main concepts, issues, and applications of mobile and intelligent agent research. Although agents are being used to develop computer systems, there is still a lack of intelligent and mobile agents being used in electronic commerce. This section focuses on the application areas and techniques used within electronic commerce. In particular, existing electronic auction systems, multi-agent markets, and electronic negotiation are investigated with a view to designing a novel Internet Auction System (IAS). For an overview of electronic markets and exchanges, see [FRI93]. This gives an interesting summary of many electronic markets (Web and non-Web) according to auction rules and practice[4]. However, figure 2.1 represents the various research themes sketched in this section. The white boxes represent areas that are important to this thesis. The other boxes represent areas of interest. These give a wider picture of the research or represent areas that are beyond the scope of this thesis. However, research from any area may be used to illustrate an issue or explore a wider concept.

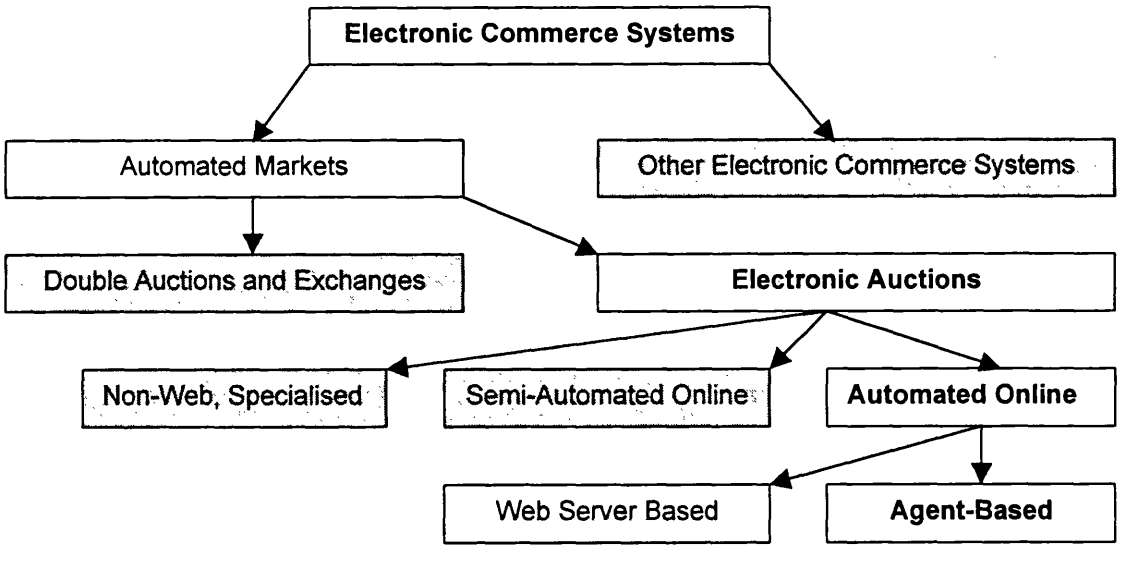


Figure 2.1 Electronic commerce research areas

[4] There are electronic auctions which are not Web-based; these tend to be specialised [FRIE93]. Some examples are the Dutch flower auction at the Teleflower auction by East African Flower Import Organisation [TFA]. They have a PC based electronic auction system for Dutch auctions. Another example is the remote auction system by the commercial company OES Inc. [OES]. This is a graphically based Dutch auction showing 'countdown clocks' on each of the bidder's PC(s). Bidders use their PC(s) to interact at the auction. Another example is the electronic auction system used by the Arizona Stock exchange [AS].

2.2.1 Semi-automated Online Auctions

Semi-automated online auctions are usually set up and run by companies that want to sell goods over the Internet. Users who want to bid must first find the auction web site. This is becoming easier with auction directories. These list auction web sites by the type of goods they auction. The Internet Auction List is a web site that gives hyper-links to over 1500 auction web sites [AUC]. Potential bidders can search for auctions selling specific goods by using its search engine. The search engine returns all the auction web sites selling that type of good. If a bidder finds an acceptable auction, he can submit bids using e-mail or web forms. Most Web-based auctions used the English method (although, there a few Dutch online auctions) and last days.

One of the most popular online auction houses is ONSALE [ONS]. They auction computers and electronic goods using the English method. Another popular online auction house is First Auction. They auction a variety of goods from cameras to golf clubs. However, some are specialised. For example, the auction house, Numismatists auctions coins. Winebid.com, as the name suggests, auctions wine and Klik-Klok [KLK] sells gold jewellery using the Dutch method. See [AUC] and [AS] for details on these and other online auctions.

All the online auctions mentioned are semi-automated because they do not enable users to set up auctions and sell over the Web. Users cannot sell goods directly; they can only buy goods being offered by the online auction house. However, some electronic auctions systems enable the user to configure the auctions themselves so that they can have more control over the auction. These automated auctions are described next.

2.2.2 Automated Online Auctions

There are fewer examples of automated auctions. These enable sellers to sell their own goods at a company's web site or set up auctions on their own web sites. Bidders still bid for goods in the same way as before using web forms. A few examples are Arizona WWWeb Works [ARZ], eBay's AuctionWeb [EBA], Webalog [GOR97], Michigan Internet AuctionBot [WU98a,], [WU98b], [WEL98], [MICH] and Infomar fish auction project, part of an Esprit funded programme [INF].

- Arizona WWWeb Works enables the seller (user) to set up an auction on the Web. Once the auction has been set up and started, bidders (other users) may bid without further intervention from the seller. A seller sets up a web page describing the item for sale (lot), minimum acceptable bid (reserve price) and an auction end date. Once this web page is set

up, the seller waits for bids to be submitted. All bids are displayed in the web page. This will continue until the auction ends at the specified date and time. Occasionally, bidders may want to ask about the lot. They can send questions using web forms. When the auction ends, the highest bidder is contacted and the deal struck. Other features include minimum bid increment (implying it uses the English type of auction). Bidders may specify an initial bid and a maximum bid. The maximum bid is kept hidden from every other bidder. The winning bidder only pays the current highest bid. This never exceeds his maximum bid. This avoids multiple bidders having to keep accepting the current asking bid. The auction system has bidder log in, registration, and comprehensive bid logs.

- AuctionWeb by eBays is one of the largest automated auction sites. It enables sellers to place their advertisements at its auction web site. Auctions last from three to seven days. After the auction has finished, both the seller and highest bidder are notified by e-mail. It is up to them to complete the transaction. Similarly to Arizona WWWebs auctions, bidders may specify a maximum bid. Also, sellers may specify a reserve price. This is not disclosed to the bidders. One difference to the Arizona WWWeb Works auction system is that private auctions may be conducted keeping everyone anonymous. Another difference is that sellers may choose English or Dutch type auctions to sell their goods.

- Webalog is a tool for creating online auctions. Its functionality is similar to the last two systems. However, they give more detail concerning how their auction system works. They automate the auction process using specially adapted web pages. They have made their web pages more interactive by embedding special (TCL) commands within the Hypertext Mark-up Language (HTML). They believe that their auction system is an improvement on current Web-based auction sites because they have created TCL commands tailor-made for automating auctions. However, Bidders still submit bids using these modified web forms.

- The Michigan Internet AuctionBot is currently the most comprehensive auction system. At the University of Michigan, they are developing an auction system that is neutral, open, private, and highly configurable. They define a system neutral as one that is not designed to make profit. In other words, bidders and sellers do not have to pay to use the system. In fact, most online auction sites do charge bidders or sellers implicitly. They define a private system as one that keeps as much information private as possible. Bids are personal and some bidders would like to be anonymous. They define an open system as one that anyone can participate in the auction. They do not want to restrict the number of bidders or sellers. Lastly, they define a highly configurable system as one that has many parameters that can be used to configure the auctions. In particular, the auction method can be set. They categorise

auctions into four types, Mth-price, (M+1)st-price, Continuous Double and Chronological Match. These are multiple seller auctions all selling the same goods. In the Mth-price auction, the clearing price is Mth highest sell or buy bid. Analogously, for the (M+1)st-price. However, in this thesis only one seller exists per auction, i.e. M=1. In this special case, the Mth-price auction becomes the English auction. The (M+1)st-price auction becomes the Vickrey auction. For details on their definitions and the other types of auction they automate, see [WU98b] and [WEL98].

- The Informar project consists of various International commercial partners (Vega Group Plc being the UK's contingent). The project aims to build a multicultural trading auction system for Fisherman, merchants and retailers for fish caught in various parts of the North Atlantic. There are two parts to the system FishCast and FishTrade. FishCast is an information service. Past auction prices, catch reports and participating fish vessels can be accessed by the parties concerned. FishTrade is the real-time Web-based auction market. Using satellite communications and Web technology they aim to allow fishermen to participate in auctions whilst still at sea. The web interface will be multilingual and handle various currencies. The project is still in its feasibility stage and no results have been published to date.

The last five examples briefly covered some automated auctions. All used web forms to interface with the auction system and server-side programs to process the bids. However, none were agent-based auction systems in which agents could bid and sell on the users' behalf. The next two sections look specifically at agent-based electronic markets and auctions.

2.2.3 Multi-Agent Markets

As with most agent fields, agent-based artificial markets is an active one. A good reference on electronic markets and the various techniques used to automate them is given by Gibney in [MKT]. This lists further hyper-links and references to 'economically inspired' computer systems. Some of the relevant centres listed in [MKT] are:

- Artificial Markets Projects, MIT;
- Autonomous Agent Group, MIT;
- Multi-agent Auction Protocols, University of Washington;
- Market Architectures, University of Minneapolis;
- Centre for Economic Learning and Social Evolution, University College London;
- DAI Research Unit, Queen Mary and Westfield College, London;
- Kasbah, Media Labs, MIT;

- Fishmarket, Artificial Intelligence Research Institute (IIIA), Spain (see next section).

Invariably, all the above centres want to create artificial markets where agents can meet to trade either as a buyer, seller, or both. In particular, they are looking at three particular problems associated with artificial markets. Firstly, the behaviour of markets with large number of interacting agents under different trading rules. Real markets form complex dynamic systems. Using simulations, they can attempt to examine specific market properties, e.g. stability, volatility, and growth. Some centres are investigating how trading rules affect artificial and real markets. They are trying to develop simulations that seamlessly integrate real trading with agent trading. Secondly, they are looking at how multi-agents can learn and adapt within these markets. A range of techniques is being used. Many of these have been inspired by biological systems, e.g. Genetic Algorithms. These will be examined in the next section. Thirdly, the problems of how to model complex financial markets. Generally, the results obtained are only as good as the market model, data sets, and inference techniques used. A few examples are:

- The Artificial Markets Group a MIT have designed a WebMarket that allows agent traders and real traders to interact seamlessly and anonymously over the Web. If real traders knew they were dealing with agents, they may behave differently. By doing this, they can explore market behaviour without real traders acting differently.

- Another project being pursued at MIT but at their Media Lab is Kasbah [CHA96]. Kasbah is an agent-based market in which users can create autonomous agents that buy and sell goods on their behalf. They are particularly interested in the interaction and competition of agents within artificial markets. In Kasbah, agents have simple buying and selling rules and do not learn. Seller agents can be configured with a desired date to sell the item by, a desired price, and a lowest acceptable price. Their negotiation strategy is to reduce their asking price over time if buyers do not accept. They used various asking bid decay strategies, e.g. decaying linearly, quadratic, and cubic. Buyer agents have similar configurable parameters except they have a highest acceptable price and their negotiation strategies increase amounts bid with time. Before any transaction is formalised, agents seek permission from their user to proceed. Kasbah agents are not mobile. They talk to each other using various methods and parameters, e.g. what-is-price?(agent, from-agent), what-is-item?(agent, from-agent) and accept-offer?(agent, from-agent, offer). Only one agent can talk with another agent at any one time. To regulate the conversations and be fair to the agents, a scheduling algorithm is used. During an agent's time slice, it may talk with another agent or re-assess its asking price or amount bid. It then has to talk to every other agent in turn. Once it has done this, it picks the most promising agents, i.e. ones with the best prices. Once an agent has decided which agent

to trade with, it contacts the agent in question and attempts to strike a deal. However, agents did irrational things like accept the first good offer when a better one was just around the corner. The designers recognised that their agents needed to be smarter and, if possible, learn from their mistakes.

- Sandholm has experimented with a Vickrey auction multi-agent system [SAN97]. He investigated the pros and cons of having agents trading for resources using a Vickrey auction. Collins et al [COLI97], [COLI98] have been looking at multi-agent protocols for artificial markets. They are particularly interested in agent negotiation, fraud prevention, and counter-speculation. Another, artificial market developed by Eriksson [ERI97] enables users and agents to coexist seamlessly. Finally, Galiano and Fraser [GAL97] are designing agent-based markets that use various protocols. They are comparing different protocols (auction, barter, single action etc.) to see how efficient they are at allocating resources. They conclude that agents that bid for resources in auctions held by the operating system are more efficient in allocating resources than centralised controllers, schedulers, and priority schemes.

Other related work in multi-agent systems has been carried out by the Distributed Artificial Intelligence group at Queen Mary and Westfield College, London University. They have designed an ADEPT (Advance Decision Environment for Process Tasks) system that models business processes using autonomous negotiating agents. Though this is primarily a co-operating multi-agent system, some of the implementation problems are shared by competitive agent systems. Further details on this and its implementation problems can be found in the [BIN97].

2.2.4 Multi-Agent Auctions

There appear to be very few working agent-based auction systems. Only three such systems claimed that they were agent-based. These are:

- Fishmarket, IIIA, Spanish Council for Scientific Research, [FIS98], [NOR97];
- Living agents from Living Systems [LIV];
- SmartBidder from Hobby Markets Online [HOB].

Out of the three systems, only Fishmarket appears to be a full-scale agent-based auction system. The Fishmarket project is primarily concerned with the dialogical aspects of intelligent agent communications and agent accountability. They are investigating a 'layered model' whereby agents share a common ontology. In other words, they want their agents to have a vocabulary

consistent and accurate for fish auctions[5]. To do this they have built a network-based market that represents a fish market. Within their artificial market, agents negotiate as buyers and sellers. In designing Fishmarket, they address two broad issues. Firstly, how do agents contend with the diversity of goods, trading conventions and timing of bids? Secondly, how do agents trade and negotiate in an effective way? For the most part, agent communications and dialogue is ignored in this thesis because unlike Fishmarket (and currently all other auction systems), the proposed IAS will automate auctions using mobile agents. Using mobile agents in a distributed electronic auction enables the system to be engineered differently because agents can meet locally to participate in auctions. Controlling local multithreaded auctions would be simpler because agent negotiation (bidding and selling) could be mediated by local auctioneer monitors. There is no need for an agent communication language or negotiation protocol. Another problem encountered by their system, is the time delay with agents communicating [BIN97]. If one agent gets information before another then this could give it an unfair advantage. This highlights another advantage of the proposed IAS since all agents would be synchronised during the auctions. Besides the linguistic aspects of their research, they are investigating and developing protocols for various auctions, e.g. Sealed-bid, English and Dutch. They are also developing Java-based 'nomadic' applet interfaces. They hope to make their system easier to use, fair and consistent to all potential buyers and sellers.

The Living Agents application is a visually based auction system. The company has released very few details. They say that their agents have the follow properties: communication and co-operation with other agents, autonomy, sensitivity and responsive to the environment and platform independent mobility. They also say that they use the following standards: Java, Object serialisation, KQML, and KIF. The only novel detail they have given about their auction system is that the whole auction process can be watched in 3D animation using a special screen.

The latest addition to the commercial based auction systems is SmartBidder by Hobby Markets Online. As with most commercial systems very few details have been released. They say that their agent bids on your behalf. It is being used at the Numismatist Online Web-based auction. It appears their agent is a simple programme. It automatically bids for you increasing its bids up to a maximum. They have used the term 'intelligent agent' very loosely.

2.2.5 Electronic Negotiation

This section gives some background to electronic negotiation. This is sub-divided into ontological (not a part of this thesis) and strategic (which is). Figure 2.2 represents the various

5 For their inspiration, they visited local Spanish fish auctions.

areas being investigated within electronic negotiation. It is based on the categorisation described by Beam and Segev [BEA96a], [BEA96b]. The white boxes represent areas that are important to this thesis. The other boxes represent areas of interest. These give a wider picture of the research or represent areas that are beyond the scope of this thesis.

Electronic negotiation concerns how agents communicate with each other, what they say, and how they say it. Ontology deals with agent vocabulary and semantics. When agents talk to each other, they must use the same language in the same way to avoid ambiguities and agents misunderstanding each other. One such problem is how an agent describes an item up for auction. If agents use different phrases for the same item, they may not understand each other. Therefore, it is essential in an artificial market to eliminate such ambiguities. However, in this thesis, items have precise names. This is a simplification but avoids complex semantic and ontological issues.

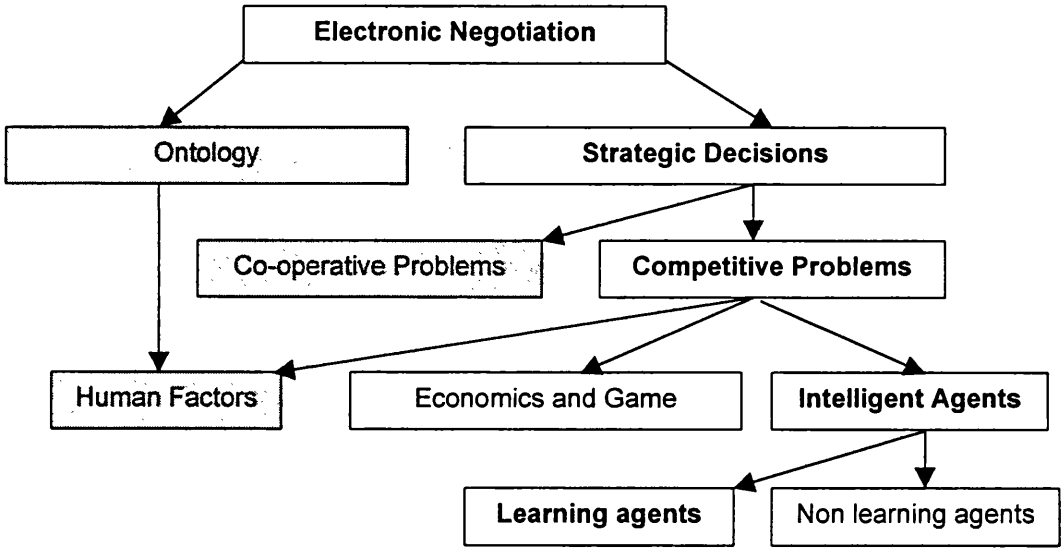


Figure 2.2 Electronic negotiation research areas

The strategic aspect of electronic negotiation concerns how agents make decisions during negotiation. Many strategies rely on secret or private information. These can be simple or complex. However, Bean and Segev [BEA96a], [BEA96b] recognise the advantages of auction based markets because as they claim, auctions are 'inference-proof'. Knowing a seller's pricing strategy does not matter in an English auction. If bidders do not collude, the seller gets the highest bidder's bid. They may accept or reject this. The point is that the outcome only depends on the competing bidders. Therefore, agent-based auction systems are a fair way to buy and sell in open markets. Bean and Segev then divided strategic negotiation into two types, depending on whether the problems are co-operative or competitive. They also divided the approaches used to solve these problems. They considered three approaches: 'Human Factors', 'Economics/Game

Theory' and 'Computer Science/Intelligent Agents'. In this thesis, only competitive problems using the 'Intelligent Agent' approach are considered[6]. However, some results from auction theory are used to check results obtained using intelligent agents. This is especially relevant to assessing how well agent strategies perform in the proposed auction system.

Beam and Segev go on to sub-categorise intelligent agents into non-learning and learning agents. They give a few examples of electronic negotiation using agents. For example, Chavez and Maes developed Kasbah using the non-learning agent approach. Also, Sandholm and Lesser investigated various market protocols for efficient resource allocation. Other groups are using learning agents to improve their ability to negotiate. Invariably, ideas from biology continue to influence the strategy-based agent research for economic and game theory problems. De Vany gives a good reference list of research in Adaptation, Dynamics, and Economic Organisation in agent systems [ADE]. He covers cellular automata, neural networks, non-linear (chaotic) systems, Genetic Algorithms (GA), and morphogenesis in his list of techniques. In particular, GA(s) have provided some goods methods to evolve agent strategies. For example, Axelrod's seminal work on evolutionary computation applied to the prisoners' dilemma problem showed these techniques are versatile [AXE81]. He found that using GA(s) to evolve agent strategies in specific environments often only produced strategies that performed well in the specific environment and poorly in others. More recently, Dworman, Kimborough and Laing [DW95], [DW96a], [DW96b] used GA(s) to evolve negotiation strategies. Oliver [OLI97a], [OLI97b] has also used GA(s) to evolve negotiation strategies for business applications. Although, they use two and three agent interactions as opposed to a 'many agent' system. So far, they have expressed positive results for complex negotiations between agents.

However, not all researchers share the optimism that these biological techniques are effective in analysing complex systems by computer simulation. Huberman casts doubt on the ability of cellular automata simulations to reflect the underlying dynamics of co-operation in social systems. He goes on to show results in co-operating systems are different when simulations are discrete to when continuous [HUB93].

2.3 Summary and Conclusion

This chapter introduced some agent technologies. In particular, mobile and intelligent agents were investigated. It was found that most mobile agent-based systems use their own script languages. None were found to use SGML as a basis for scripting and defining agents. This would provide a good way to bring agents within the umbrella of Web technology i.e. HTML

6 Agents in the proposed IAS may co-operate to find auctions but not during bidding or selling.

web pages and agents are made more analogous. Moreover, none of the mobile agent infrastructures investigated used HTTP and sockets as a basis for constructing a mobile agent infrastructure. Next, the current techniques and application of intelligent agents were investigated. There were few details describing the precise nature of an agent's intelligence, especially in commercial system. In particular, it was found that few agents, if any, used embedded fuzzy-genetic adaptable strategies. Therefore, designing AMA(s) using these techniques would provide novel agents that could be use to design multi-agent systems.

Next, online auctions were investigated. It was found that most Internet or Web-based auctions use web forms. Very few electronic auctions were agent-based. The most comprehensive and sophisticated agent-based electronic auction was Fishmarket. However, they concentrated on the ontological issues of electronic negotiation. Their agents were not mobile either. It appears there were no electronic auctions using mobile agents. Given that mobile agents enable systems to be designed differently, they may prove to be more effective at creating a distributed auction system than current systems using immobile agents.

The final sections of this chapter described various themes found in electronic commerce. This included ontology and strategic decision making for co-operative and competitive problems. Beam and Segev suggest three main approaches to tackle these problems. The 'Human Factors' approach (the more subjective side to negotiation, i.e. cultural and psychological) was ignored. The 'Economic and Game Theoretical' approach (e.g. negotiation is seen as a game between various players) would be used to verify an agent's performance. The 'Intelligent Agent' approach would be used to implement electronic negotiation in Internet auctions, although, the effectiveness of AI techniques in agent negotiations, according to the research, was inconclusive. Therefore, this thesis intends to use fuzzy logic and GA(s) to encode and evolve agent strategies. In particular, fuzzy logic is used to partition agents' information spaces, genetic structures are used to encode their strategy spaces and Fuzzy-Genetic Algorithms are used to evolve them.

Given that the proposed AMA and the AMA-based IAS appear to be novel. The following two chapters describe how the AMA could be implemented. In particular, Chapter 3 discusses issues relating to agent mobility and management and Chapter 4 discusses issues relating to agent automation and adaptation. Furthermore, these AMA(s) are used to create a fully automated IAS. In this IAS, adaptable (immobile) Seller agents automate selling and adaptable mobile Bidder agents automate the searching and bidding in remote auctions held by Seller agents. In particular, Chapter 5 describes the IAS architecture. Chapter 6 describes the Seller agent and Chapter 7 describes the Bidder agent.

Chapter 3 – Creating and Managing Mobile Agents

The next two chapters discuss the development of Adaptable Mobile Agents (AMA) that could be used to automate electronic commerce systems in a novel way. However, this chapter focuses on tools and concepts associated with agent mobility and management (the next chapter focuses on agent automation and adaptation). Within this chapter, the first few sections describe a mobile agent infrastructure that is Web-based, dynamic and supports mobile users. In addition, agent filtering, dynamic routing, and user addressing schemes are covered. The remaining sections describe how users manage agents by creating, monitoring, saving and loading them into their agent-based systems. The last section summarises the achievements made.

3.1 Mobile Agent Infrastructure

Systems that use mobile agents have benefits [NWA96]. Firstly, mobile agents can move to remote computers and communicate locally. This would reduce network traffic between computers in a distributed system. Secondly, they can make agents easier to co-ordinate using agent monitors that synchronise the concurrently running agent threads. Thirdly, they can make inter-agent communications simpler again using agent monitors. Fourthly, they can enhance asynchronous computing and provide a new flexible way to model distributed systems. These benefits will become clearer during the development of the IAS in Chapter 5. However, as mentioned in the last chapter, there are drawbacks. Security is considered the main reason why there is a resistance to adopt mobile agents systems. Although security is not ignored in designing the mobile agent system, it is not a central issue and provides an area for further work, see Section 9.3.

3.1.1 Requirements

Mobile agents require an infrastructure. In this thesis, a mobile agent infrastructure is a network of computers running software that enables agents to move from one computer to another. Infrastructures can be designed with specific requirements. The main requirements considered important for this mobile agent system are listed below. These are explained and where necessary some reasons for adopting the requirement are given.

- Platform Independent/Open System
- Web-based Technologies

- Distributed Client/Server Networks
- Dynamic and Transient Networks
- Mobile User Support
- Rudimentary Security

Platform Independent/Open System

Mobile agents should be able to move from one computer to the next within the agent infrastructure and not be limited to computers running one particular operating system. For example, a mobile agent may move from a computer using UNIX to another using Windows NT. It could then move to another computer running Windows 95 or 98. A mobile agent infrastructure that enables agents to run and move to computers with different operating systems is a *Platform Independent/Open system.*

Web-based Technologies

The World Wide Web (Web) is a global network of web servers [CRO96]. *Web technologies* primarily concern the interaction and functionality of web servers and web clients (browsers). Briefly, a browser is a program used to access information held on web servers. It uses the protocol, Hypertext Transfer Protocol (HTTP) to communicate with the web servers. The web servers hold web pages in the form of hypertext information. A browser can download web pages by sending HTTP requests to the web server. The web pages are written or scripted in a language called Hypertext Mark up Language, (HTML). The browser decodes the HTML of the web page and displays it graphically. In addition, web servers can communicate with other computers using Common Gateway Interfaces (CGI), i.e. computer programs [GUN96]. There are two main reasons why the mobile agent infrastructure should use Web technologies. Firstly, these technologies are popular and have proved to be relatively simple and effective. The global network of web servers could be used as a backbone for a mobile agent infrastructure. Secondly, the HTTP protocol has many useful features that could be adopted, e.g. security. This would prevent reinventing features that have already proved effective.

Distributed Client/Server Networks

The *Distributed Client/Server* architectural paradigm is very useful for modelling computer systems, see [ORF96]. In this thesis, a mobile agent infrastructure consists of a network of

computers. Each computer in the network can act as an agent client/server[1] or web server by running appropriate software (the reasons for having web servers as well as agent servers will be explained latter). An agent moves from a computer acting as a client to another computer acting as a server. The agent server accepts the agent whilst listening on its dedicated port and allocates it to a shared-variable. The contents of this shared-variable are passed to the agent client as a string and run locally. This cycle continues as the agent moves around the network from computer to computer. However, a mobile agent infrastructure should be arbitrarily complex. It should consist of any number of agent client/servers and web servers. In addition, if the mobile agent infrastructure straddles two computer networks then it may require a proxy server or firewall. A firewall is a computer that performs security checks on information (in this case, mobile agents) passing from one network to another. In other words, it creates a wall between an internal network of computers and the outside external network, e.g. Internet. For example, agents can be filtered and checked by the firewall. Moreover, a firewall can prevent agents from leaving the internal network and vice versa, the firewall can prevent agents on the outside from gaining entry into the internal network.

Figures 3.1 and 3.2 show two simple infrastructures for mobile agents. The first infrastructure shows multiple agent client/servers on one computer. To have an agent network there must be at least two agent client/servers. The second infrastructure shows two computers (could be two or more) each running one agent client/server.

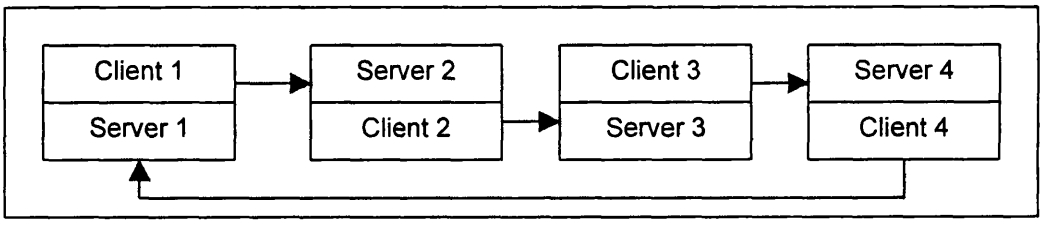


Figure 3.1 Computer running multiple agent client/servers

Mobile agents should be able to move from any agent client to any agent server. However, after they have moved from client-x to server-y they are always passed to client-y as a string ready to move to another server. They cannot move from client to client or server to server. Figure 3.1 shows an agent moving from client-1 to server-2, from client-2 to server-3, from client-3 to server-4 and from client-4 back to server-1.

[1] In fact, it can run multiple clients and multiple servers by using ports. Therefore, a mobile agent infrastructure can be physically located on one computer; it does not have to be distributed. An agent can move from a client to a server on the same computer. How clients and servers are uniquely specified on the same computer is discussed in the Section, Joining and Leaving an Agent Network.

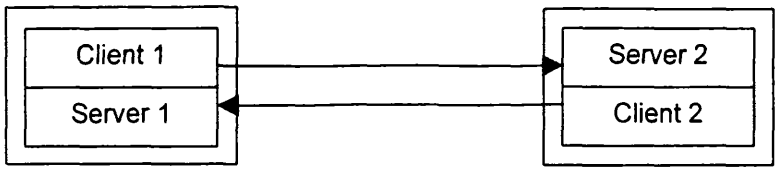


Figure 3.2 Two computers each running one agent client/server

Figure 3.2 shows how two computers form the agent network. The arrows represent a mobile agent moving from one computer to the next as its passes from client to server. A network becomes difficult to manage when the number of computers running agent client/servers, web servers or firewalls increases (further details are given in the Section 3.1.2).

Dynamic and Transient Networks

In this thesis, a mobile agent infrastructure that is capable of growing or shrinking at any time is called *Dynamic*; i.e. the number of agent client/servers in the agent network may increase or decrease. Users should be able to join the network by logging on from their computers. They may also leave the network by logging off. Users joining and leaving should not affect other users. The agent network should be able to start with no users, grow to any number of logged-in users, and reduce back to no users as they log out. In other words, the agent network is *transient*; users can completely shut it down and restart it by logging in and re-creating the agent network.

Mobile User Support

To enhance the mobile agent infrastructure, it should support *mobile users*. This should enable users to join (log into) an agent network from one computer, leave the agent network (log out) and then rejoin (log in) from a different computer. This would give users more flexibility because they are not restricted to joining the agent network from one particular computer.

Rudimentary Security

For the most part, computer security is a complex and difficult area [GAR94]. However, the mobile agent infrastructure is required to have some *rudimentary security*. In particular, every user must log in to join an agent network and must log out to leave it. During log-in, a user must supply a password. Another security requirement is to filter agents. This enables users to filter and check all agents that move from one computer in the network to their computer (agent client/server). As suggested earlier, the mobile agent infrastructure may also use a firewall to

protect the agent network or more sophisticated security features like encryption and authentication [CHE96]. In addition, by using HTTP some additional security measures may be implemented [BER94]. However, for the most part security is ignored.

This completes the main requirements for the mobile agent system. In particular, the mobile agent infrastructure should be platform independent, Web-based, distributed, dynamic and transient, supportive of mobile users and have rudimentary security. However, there are other important issues, for example, system performance and interoperability [ALM96], [BI96]. System performance concerns running large numbers of agents to see if the system is inefficient, causes delays or even crashes. This will be investigated in Chapter 8. Interoperability concerns agents moving from one infrastructure to another. Admittedly, the infrastructure has been designed in isolation and this complex issue is left to further work.

3.1.2 Design Solutions

This section describes how the mobile agent infrastructure is implemented to meet the requirements. The design solutions are described in four main sections, namely, Agent Networks, Joining and Leaving an Agent Network, Security and Infrastructure. The section, Infrastructure covers sections on posting agents over the Web, routing agents, and messaging protocols.

Agent Networks

In this thesis, agent networks are defined as transient and dynamic networks of users running agent client/servers that enable users to create and run mobile agents. These mobile agents may move from user to user within the agent network. Moreover, the part of the agent that is mobile consists of a text-based (ASCII) script written in some agent language and not an object or object code [FRA]. Therefore, agents and agent scripts are used synonymously in this and latter chapters.

To implement the mobile agent infrastructure, users must have various languages, software, and protocols installed on their computers. These are the Java Development Kit (JDK)[2] [COH96], [FLA96], Perl language [WAL92], web server software [SPA96], Internet protocols and software (TCP/IP and sockets) and an Internet connection. Java is platform independent language from

2 Briefly, there are three components to Java: the Java Virtual Machine (JVM), the Java compiler, and the Java interpreter. This makes up the JDK. The Java Compiler parses source code and outputs Java byte codes. These are interpreted using the Java Interpreter. The JVM converts interpreted byte codes to machine instructions. In other words, the JVM sits between the computer's operating system/hardware platform and the Java running program. Each type of operating system/hardware platform requires a different JVM. However, a Java program can run on any computer with an appropriate JVM.

Sun Microsystems. This is why Java was adopted as the main implementation language. It enables agent client/servers to run on computers with different operating systems, currently: UNIX, Windows 95 and NT, and Macintosh. Similarly, web servers and the Perl language are available for different platforms. This would make the mobile agent infrastructure very flexible because it could include computers running different operating systems.

In order to introduce the agent network design a simple example is given. The network in figure 3.3 shows an agent moving from one user's computer (source) to another user's computer (destination) via an intermediate web server. Every time an agent passes through a web server, a resident CGI program written in Perl (agent router) is executed. This program accepts agents and routes them to their final destinations, i.e. other users. This process is described in six steps:

1. Agent runs at source, it instructs the agent client to make an HTTP post request to destination's web server (may or may not be running on the same computer).
2. Agent router on web server checks destination exists and is running.
3. If destination exists and is running then the agent is posted to web server.
4. Web server receives agent and redirects it to the agent router program.
5. Agent router forms a socket connection with the destination's agent server port.
6. Agent router sends the agent via the socket to destination's agent server, which passes it to the corresponding agent client as a string. The agent then runs locally and the socket connection is closed.

Figure 3.3 Agent moving from one user to another

The web server is an important part of the agent network for three reasons. Firstly, they enable users to be mobile. As long as the web server's agent router has a user's current address, an agent can be routed to the mobile user. Secondly, web servers already form a large global network (the Web). Mobile agents should exploit this global network. Thirdly, web servers already have security features that mobile agent systems could use as a first defense against unwanted agents. Exactly how mobile users are given unique addresses in the agent network so that agents can be routed to them, and how mobile users join and leave the agent network is now discussed.

Joining and Leaving an Agent Network

Every user requires a computer connected to the Internet to join the mobile agent network. Users interact with the mobile agent system using Graphical User Interfaces (GUI) [3]. To join, a user must log in. On logging in, a user must supply various pieces of information. This includes addressing information. A user's address consists of seven parts, see the following table 3.1.

Username	Username in the agent system
Password	Security Check
Computer's Address	Computer's unique address, Domain Name or Internet Protocol (IP) address
Agent Server's Port[4]	Required by the agent server to listen for incoming agents and messages
Agent Router's relative address on Web Server	This is the user's web server IP address and the CGI directory location of its agent router program for routing incoming agents.
Proxy Server's IP or DNS address	If the user's computer is not part of an Intranet then no proxy server address is required.
Proxy Server's HTTP Services Port	Again not required if the user's computer is not part of an Intranet. Otherwise it is generally set to 8000, the special port for HTTP requests.

Table 3.1 Information required by a user to log into the agent system

Users do not have to enter all the details every time they log in. Usually they would only enter their *usernames* and *passwords*. The remaining details can be read from a configuration file set up by a systems administrator. In particular, the administrator would set up each user's *agent server port* so that they were unique. They would configure the *proxy server's IP address* and *HTTP services port*. In addition, they would set up the web server and its resident CGI agent router. The *agent router relative address* should be fixed for a particular agent infrastructure. However, the location of the agent router program within a CGI file directory is not important. A user's Username, Computer's Address and Agent Server's Port defines the user's unique address. However, agents do not need to know this full address. All they need to know is the user's

3 These GUI(s) are similar in style to a web browser. However, a web browser using Java applets was not used as the system's interface because the restrictions placed on Java applets. These prevented Java applets from forming connections to remote computers or reading/writing to local files. Therefore, they were not suitable for this mobile agent system. However, newer versions of Java have introduced trusted applets that posses digital signatures that once authenticated relax their security restrictions.

4 Users running their agent servers on the same computer system must use different ports to enable them to be identified uniquely.

User*name* and *Web Server's Address*. This is because usernames and web server addresses in the agent system remain fixed; the computers they log in from do not. In other words, users can be mobile. They may join the agent network from different computers. However, they must be registered at one fixed addressed web server. Every time a user logs into an agent network from a computer, the following user details are registered with their fixed address web server[5]:

- Name;
- Password;
- Computer Address;
- Agent Server's Port;
- Log-in Status;
- Agent Filter Option.

This list is stored as a local file on the web server (Username file) and every time a user logs in or out, their details are updated by the web server's agent router program. This is useful for three reasons. Firstly, it provides a mechanism for dynamic routing and mobile user support; i.e. users are not restricted to one computer location. Secondly, it can enforce security, i.e. if an impostor attempts to log on whilst the user is already logged on, the system will abort their log-in because only one log-in is permitted per user. Thirdly, it enables agents to know whether the user exists, is logged on and where the user is currently located. This process is described in figure 3.4.

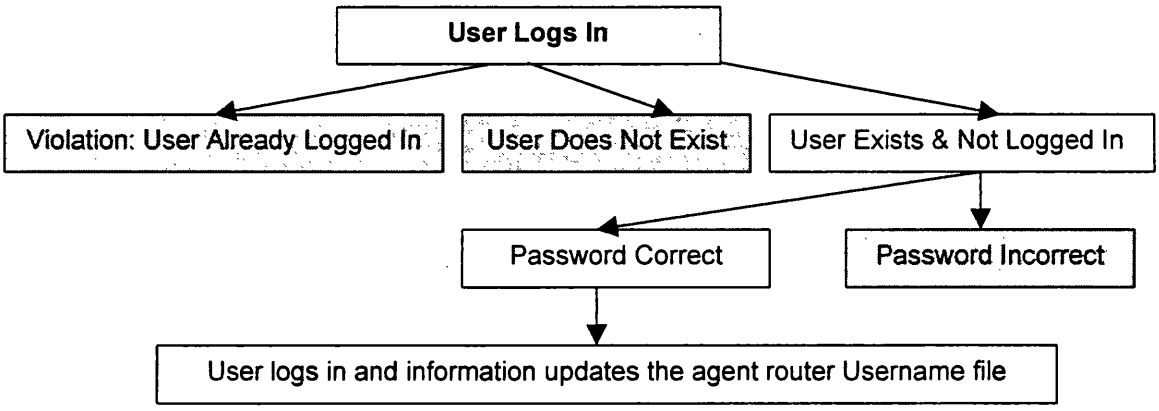


Figure 3.4 User log-in outcomes

5 Many users may be registered with one particular web server. Therefore, every web server stores a list of its registered users.

Agent Security

In this mobile agent system, agent security is rudimentary. There are many security problems; some originating from the computer systems[6], some from the networks [WWW] and some from the application itself[7] that need to be addressed. In particular, users may filter incoming agents, see figure 3.5.

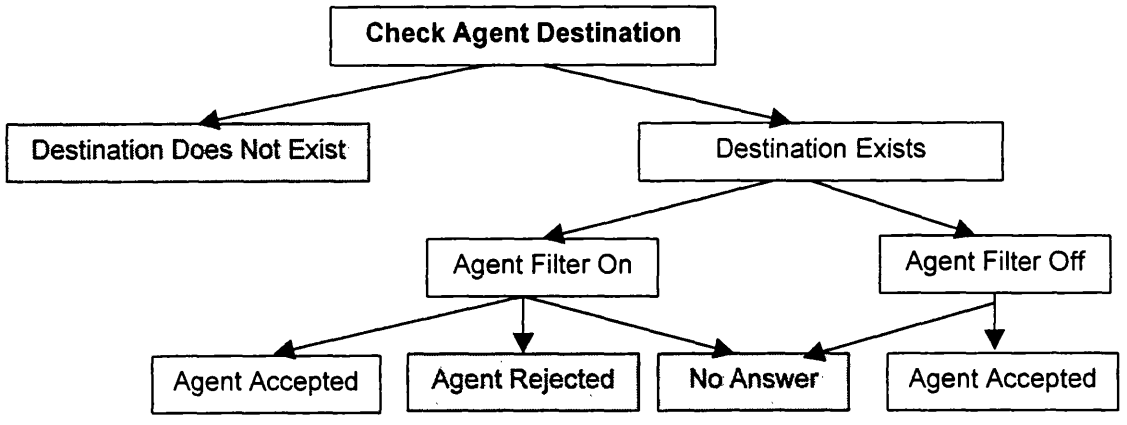


Figure 3.5 Agent filtering

If a user has agent filtering activated then every agent attempting to visit the user triggers a warning. A GUI window pops up and asks the user if they accept the agent. If the user accepts, the agent is sent an 'accept' message. The agent is then allowed to move to the user's computer. If the user does not accept, the agent is sent a 'reject' message and the agent cannot move to the user. A final scenario is that the user is absent from their computer. The warning window waits for a limited time. If the user does not respond the window automatically closes and sends a 'not available' message. This also prevents the agent from moving to the destination. However, there are serious security problems. Firstly, agent scripts are readable ASCII text. They clearly need to

---

6 Other security breaches can occur using CGI programs [CGI]. On starting a web server in UNIX, its daemon process, HTTPd has 'root' status. This allows the process to be bound to port 80 (a trusted port) so that it can read/write to system log files. However, when a user sends a request to a web server (and executes a CGI program), it spawns a child process. These have lower status and fewer permissions (in UNIX, they have 'nobody' status). They can only execute files with the same low status. However, in this agent system, agent router programs are required to form socket connections to other processes and write to local files. To do this, the agent router requires higher permissions than a 'nobody' process would have. Therefore, programs are given higher permissions so that they can form socket connections. In UNIX, the agent router is given higher permissions, by setting SUID (super user id). The downside to this is that programs with higher permissions can potentially do harm, if infected by a virus [GAR94]. Another, problem encountered by using CGI programs written in Perl is the use of 'tainted' variables. A 'tainted' variable is a security mechanism to prevent processes from manipulating other processes, e.g. forming remote socket connections or writing to local files. A CGI program written in Perl that receives data variables from external sources become tainted, e.g. destination computer's address and port number. This prevents the agent router from using these variables to form socket connections. However, if Perl's regular expressions functions are used to extract the information from a 'tainted' variable, it becomes untainted [WAL92].

7 However, if an agent were at a particular location and the computer crashed or lost power, the agent would not be saved. The agents' final fate would be unknown to its user.

be encrypted on transmission and only deciphered when and where necessary. As mentioned earlier, HTTP was adopted because it has security features. In addition, there are secure HTTP protocols, e.g. S-HTTP that provide encryption and authentication. However, these security features have not been implemented in the mobile agent system.

Infrastructure

This section explains in more detail how agent clients/servers, web servers and routers communicate with each other as an agent moves through the network. To aid this explanation, a more complex agent network is given in the following diagram.

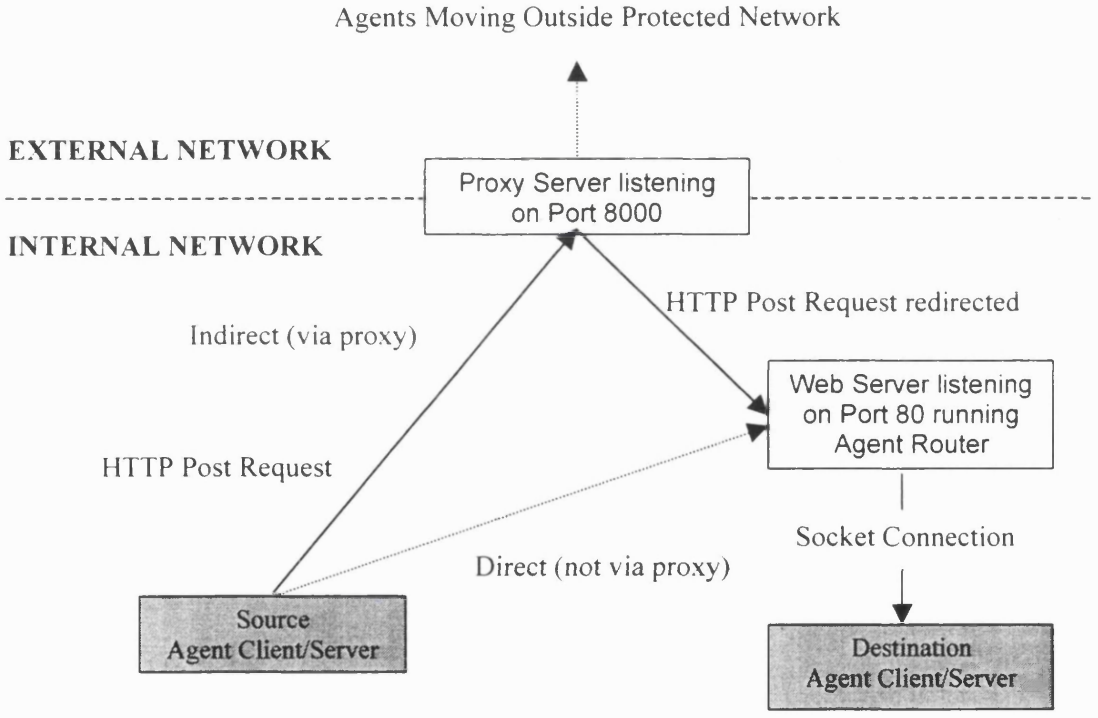


Figure 3.6 An agent network with a Proxy server

The agent network in figure 3.6 consists of four computers. One is running a web server, two are running agent client/servers and one is running a proxy server. An agent on the source computer moves to the destination via the destination's web server *and* proxy server (see solid line). However, the proxy server is used for security, creating a wall between the internal (called an Intranet) and external (the Internet) network of computers. However, the agent network may not have a proxy server. It is shown here because some networks enforce security using one. In general, agents can only move within the internal network. If they are required to move to the external network and back in again, the proxy server needs to be configured to do this. It must recognize a mobile agent and allow it free access to move into and out of internal networks.

However, if the agent network does not have a proxy server, the agent can pass directly to the destination's web server (see dotted line).

Posting Agents over the Web

Central to the mobile agent infrastructure is how the mobile agents move from one user's computer (running an agent client) to another user's computer (running an agent server). Firstly, the agent is required to move from the agent client to the destination's web server for routing. The HTTP post request is used for this purpose [LIN96]. The agent client on the source computer makes an HTTP post request [SPA96] to the proxy server (if there is one) or directly to the destination web server (called the origin server). The HTTP request takes two different forms depending on whether the agent is moving via a proxy server or directly to the origin server. If it moves through a proxy server an absolute address is used otherwise a relative address is used. These addressing schemes (Uniform Resource Locators, URL [SPA96]) take the form:

- **Absolute URL-Address** = *http://host/path/extra-path-information*
- **Relative URL-Address** =*/path/extra-path-information*

The *host* is the Internet Protocol (IP) address of the destination's web server (often has an alias called the computer's "domain name"). The *path* is the location of the agent router program on the web server, i.e. the directory. And, *extra-path-information* is the name of the agent router program. The HTTP request takes the form:

POST ***URL_Address*** *HTTP/1.0 CRLF*
Host: ***host_address*** *CRLF*
Content-length: ***N*** (some positive integer)

Data *(*consisting of the N bytes of ASCII text)

These lines represent an HTTP post request [BER94]. The first line says *POST* (send) *N* bytes of data to *URL-Address* (web server's agent router program). The second line at first appears redundant but is required if the request is made directly to an origin web server. The third line tells the web server to expect N bytes of ASCII text. A mandatory blank line is next and finally the data itself. This could contain agent commands or the agent itself. *CRLF* means 'carriage return, line feed' and forms part of the HTTP post request format. The *HTTP/1.0* part of line one after the URL-Address informs the web server of the version of HTTP being used. If the request is made via a proxy server, the *Host* request header (second line) is not required. In addition,

Proxy servers require an absolute URL-Address of the form, *http://host/path/agent_router_program_name*. However, if the request is made directly to the destination's web server, the *Host* request header is required. The URL-Address must be a relative address of the form */host/path/agent_router_program_name*, i.e. no *http:/* prefix.

Therefore, the agent client (source) sends the agent to the destination's web server using an HTTP post request as described above. This can be via a proxy server or directly to the destination's web server. The agent client forms a socket connection with the proxy server or the origin web server. Typically, for a proxy server the socket is made to port 8000 and for origin web servers to port 80. Further details on why this is the case can be found in [GAR94]. Once a connection is formed, the agent client sends the HTTP post request as a stream of ASCII bytes. If the proxy server receives the request, it repackages the request and redirects it to the destination's web server. If the request comes directly from an agent client, the web server receives the request unaltered. Once the web server receives the request, it redirects the request to its CGI program (agent router) that is located in the */path/* directory. This automatically starts the agent router program running.

Routing Agents

The agent router is a program that dynamically routes agents from one user to the next within the agent network. It stores all the current usernames and addresses in a local file. In addition, it logs which users are currently logged into the system and which users are not. This enables agents to be routed to a user's current address or informed that the user is not currently connected to the agent network. Once the web server has received an HTTP post request, it processes it. Since the HTTP request contains a URL with a CGI program name as the specific location, it redirects the trailing *Data* to the CGI program. It does this by passing the data as a stream of bytes to the CGI program as standard input [GUN96]. The CGI program accepts the input data and then processes it. In the case of the agent router, the program extracts from this data, details about where the agent wants to go. At this stage, the agent only knows who it wants go to. However, it does not know where it needs to go to find them. The agent router program checks the destination's username by looking it up in a local (Username) file listing all the registered usernames and their current addresses. Once it has found the correct address (host name and port number), it forms a socket connection with the destination's agent server. If the connection is accepted, the agent is passed to the agent server as a stream of bytes and stored in a string variable. When the agent has passed to the agent server, the socket connection is closed, see Appendix D for the agent router's source code. The agent script is now at its destination. It can be passed to the agent client as a text string ready to be run (interpreted) so that it performs tasks or moves to another user's computer running an agent client/server.

Messaging Protocols

This section specifies messages sent between clients, agent servers and web servers: see table 3.2. In this context, clients include agent clients and user clients. Agent clients are clients that agents use to communicate with the web server (e.g. to move) and user clients are the clients that users use to interact with the mobile agent system (e.g. during log-in).

	Agent Router	
	Messages Sent	**Replies**
User Client	LOGIN	USER CLEARED | MEMBER VIOLATION | INCORRECT DETAILS
	LOGOFF	LOGGED OFF
Agent Client	ACCEPTANCE	ACCEPTED | REJECTED
	AGENT	INCORRECT NAME | AGENT HAS LEFT
	UPDATE | LOG	UPDATED
	Agent Server	
	Messages Sent	**Replies**
Agent Router	CHECK SOURCE	ACCEPTED | REJECTED
	AGENT	No reply
	UPDATE | LOG	No reply

Table 3.2 Message protocol in mobile agent system

Messages are sent from one client (source) via the destination's agent web server to the agent server (destination). The web server either replies to the client or contacts the agent server by passing a message via a socket connection. The messages are described in the next few sections.

Messages from Clients to Web Server/Agent Router

The agent router accepts messages from both user clients and agent clients. It performs three functions. Firstly, it enables users to log into and out off the mobile agent network. Secondly, it

routes agents. Thirdly, it enables agents to send messages back to their users. All text messages from clients to web servers take the following form:

> ***fromName*** delimiter ***fromAddress*** delimiter ***messageType*** delimiter ***toName*** delimiter ***toAddress*** delimiter ***info1*** delimiter ***info2*** delimiter ***info3*** delimiter ***info4*** delimiter ***info5*** delimiter ***info6***

This ASCII text string is sent to the web server as an HTTP post request. Delimiters separate the sender's name and address, message type and the destination's name and address. The six pieces of information are dependent on the message type, e.g. if *messageType* is AGENT then info1 is the agent script. A typical HTTP post request (given below) is sent as ASCII byte codes to the web server.

> POST ***destination_Web_Server_Agent_Router_address*** HTTP/1.0 + CRLF
> Host: ***Web_Server_Domain_Name*** CRLF
> Content-length: ***size_of_data*** \n\n
> ***message***

The six message types sent by a client to a web server are LOGIN, LOGOFF, AGENT, UPDATE, LOG and ACCEPTANCE. When users log into the agent network, their user clients send LOGON type messages with their username, current location (computer's IP address) and password to register with their web server. This information is used by web servers to route agents to their current locations. Similarly, when users log off, a LOGOFF message is sent to their web server. This information enables the web server to inform potential agents wanting to visit them that they are no longer part of the agent network. When agents want to move to another user, their clients send to the user's web server an AGENT message together with their agent scripts. When agents want to inform their users of where they are, their clients send UPDATE messages together with their current location. When agents attempt to move to another user within an agent network, they may be required to send ACCEPTANCE messages to ask the user for permission.

Replies from Web Server/Agent Router to Clients

Once a client (source) has sent an HTTP post request message to a web server's agent router program, the program replies to the source. The following nine replies are used by the agent router to answer a client's message: USER CLEARED; MEMBER VIOLATION; INCORRECT DETAILS; INCORRECT NAME; LOGGED OFF; AGENT HAS LEFT; UPDATED;

ACCEPTED and REJECTED. When users attempt to log into the agent network, their clients can receive one of three replies from the agent router. If a user has supplied the correct information, the agent router replies with USER CLEARED. If a user is already logged into the agent network, the agent router replies with USER VIOLATION. Moreover, if a user sent an incorrect username, the agent router replies with, INCORRECT DETAILS. If a user logs off successfully, the agent router replies with, LOGGED OFF. If an agent attempts to communicate with an unknown user, the agent router replies with INCORRECT NAME. If an agent has the correct username and the destination has agent filtering activated the agent is required to ask permission to move to the destination. If the destination accepts the agent, the agent router replies with ACCEPTED otherwise it replies with REJECTED. If the agent is accepted then the agent router routes the agent to its destination's agent server. Once the agent has moved, the agent router replies with, AGENT HAS LEFT. Finally, if an agent successfully sends an UPDATE message to its user, the agent router replies with UPDATED.

Messages from Web Server/Agent Router to Agent Server

The agent router communicates with the appropriate agent server by sending messages via temporary socket connections. The six messages sent from an agent router program to a destination's agent server are AGENT; LOG; UPDATE; CHECK SOURCE; ACCEPTED and REJECTED. These occur in the following cases. Once the agent has been accepted; the agent router informs the destination's agent server that an agent is to be sent by sending an AGENT message, together with the agent script via its socket connection. Similarly, if an agent wants to update its user with information, it sends via the agent router a LOG or UPDATE message to its user's agent server. Finally, if a user has agent filtering activated; any agents attempting to move to the user's computer are checked. In this case, the agent router sends a CHECK SOURCE message together with the agent's username, current location and ID to the destination's agent server. The agent server then warns its user that an agent created by a certain user from a particular location is trying to move to their computer. The user can then either accept or reject the agent. The replies sent from the local agent server back to the agent router are discussed next.

Message replies from Agent Server to Web Server/Agent Router

There are only two messages sent back from an agent server to the agent router. If a user activates agent filtering then when an agent attempts to move to the user via the agent router, the agent must seek permission to do this. An agent seeking permission to move to a destination first informs the destination's agent router that it wants to move to the destination. The agent router forms a socket connection with the destination's agent server and sends a CHECK SOURCE

message. If the destination is happy to accept the agent, it replies with an ACCEPTED message, otherwise it replies with a REJECTED message. In turn, the agent router redirects this message to the source's agent server to inform them that the agent can or cannot move to the destination.

3.2 Managing Agents

So far in this thesis, mobile agents only move from computer to computer (they do not perform tasks, see next chapter for task-based mobile agents). However, users still need to interface with the system to manage their agents. They do this using browser-type GUI(s). Interfacing with agent systems is an interesting area in itself. Animation and adaptive user interfaces are being researched because they are more realistic ways to interface with agent-based systems, although, this is beyond the scope of this thesis [BIC98], [BOS97]. However, this section intends to give an overview of a few agent management issues that are important to this mobile agent system.

3.2.1 Requirements

The four requirements for managing agents are categorised into creating agents, agent exploration, monitoring agents and utilities. Some issues are how can users create and reuse their agents? How can agents explore their environments? How can users monitor and utilise their agents?

Agent Creation and Exploration

Users should be able to create as many agents as they wish and run them simultaneously. More specifically, users should be able to create agents in three ways. Firstly, they should be able to use GUI(s) to configure agents. Secondly, they should be able to write and edit agents using simple text editors. Thirdly, they should be able to load agents from their own computer or remote computers. As mentioned earlier, the agents discussed so far do not perform tasks but move around a mobile agent network. Agents can move around in one of two ways. Firstly, they can be programmed by a user to visit various users in a specified order (a schedule without time constraints). Secondly, the agent itself can decide where to go next; it can explore its environment. In this mobile agent system, agents should be able to do both. In addition, they should be able to navigate themselves around the agent network, coping with incorrect, unknown and logged-out user destinations.

However, one aspect of agent scheduling that is not required is to have an agent's schedule strictly dependent on time. In other words, an agent would arrive and leave at their destinations at

a specified date and time. This was disregarded because in agent-based electronic markets, users only care that their agents visit their specified markets in order, do some trading before returning home. Specifying how long they should be at a market could jeopardise trading opportunities. Particularly in electronic auctions, agents should stay until the auction is over before moving on to other auctions.

Agent Monitoring

Once users have created and run their agents, they should be able to monitor them and find out where they are and what they are doing. They should be able to track their agents in real-time. In addition, users should be able to monitor all agents running locally on their computer whether from other users, or their own. However, distributed agent systems pose problems. In particular, users and agents can congest the network by sending too many messages to each other. Flooding the agent network with messages would slow the system, too few, and users lose track of what is happening. In this agent system, all agent messages sent to their users should be simple text-based descriptions stored in a time-dependent log format. Furthermore, animators showing users what is happening in a graphical or animated way could reinterpret these messages. This provides an interesting area for further work, see Chapter 9.

However, one requirement not imposed on this agent system is to allow users to remotely control their agents. Once users have created and run their agents, they become autonomous. They move around the system and perform tasks on remote computers but they cannot be recalled or redirected. To be able to stop and start remote agents and pass them new information would be very useful. It is conceivable that users may want to stop their remote mobile agents from continuing or update their agents with new information. This is especially relevant to agents trading in electronic markets. Getting up-to-date information to agents would give them an advantage. However, as will be explained in the next chapter, if the agents possess good sets of strategies then the need for remotely controlled agents is reduced.

Agent Utilities

In addition to creating and monitoring agents, users may want to edit, parse and adapt their agents. Moreover, users may want their agents automatically loaded when they log into the agent network and automatically saved when they log off. This would help prevent agents being lost during system crashes. Lastly, a dynamic computer address book listing all the currently connected computers in the agent network would be useful.

3.2.2 Design Solutions

In this section, various design solutions for managing agents were implemented into the mobile agent system. Firstly, the design solutions for creating agents and implementing agent exploration are described. Next, the design solutions for monitoring agents are described. Finally, the design solutions for the agent utilities are described.

Agent Creation and Exploration

Users must first log into the agent system before they can create agents. Once users have entered their details using GUI(s), their details are sent to the agent router to be verified. Once their details have been cleared, they can use the GUI(s) to configure their agents. To configure a mobile agent's schedule the following three parameters are set:

- List of users (with their addresses) to visit;
- Return-by date and time;
- Explore-until date and time.

The schedule enables a user to send their mobile agent to a sequence of users in a specified order, explore the agent network and return to the user by a specified date and time. Once the agent has started, it will attempt to visit each user, and perform its tasks there. If a user's address in the schedule is incorrect, or the user is not connected to the agent system, or the user refuses the agent permission to visit then the agent will log this and bypass the address. It will then attempt to move to the next user in its schedule.

The 'return-by date and time' parameter enables a user to determine when the agent returns back to them. In fact, a user could create an agent on one computer, log off and back in again on a different computer and the agent would return home to the user's current location. Therefore, the user only has to worry about rejoining the agent system before the agent is due back.

After an agent has visited its list of users, it can explore other users within the agent network[8]. These may be unknown to the user. The mechanism works as follows. Each agent has an attached list of usernames and addresses that the user believes are currently logged into the agent system[9]. As the agent moves from one user to the next via their web servers, it must disclose its

---

8 A user can prevent an agent exploring by setting the 'explore-until date/time' parameter to 'now'.

9 A user's address is not the actual user's computer IP address but a combination of the user's own unique username and web server name. E.g. a user called, 'kyzyl' may be logged on at computer with domain

list of user addresses to the current user's computer. All agents must do this. Therefore, when an agent completes its list of users it can ask the current computer for a new address, i.e. one not in its list. The agent then moves to this new user. The agent continues to ask for new addresses until either, the 'explore-until date and time' or 'return-by date and time' has passed. An agent system that enables agents to co-operate in this way has a few advantages. Firstly, it enables them to share their knowledge of users within the agent network. This could be extended to include other important pieces of information, e.g. which users are currently logged on or network problems. Secondly, it enables agents to find users within the agent network that even the agent's creator may not have been aware of. Thirdly, it reduces the need for a central database of user addresses.

Agent Monitoring

In this mobile agent system, users can monitor their agents in three ways. Firstly, agents can send messages home, indicating where they are and what they are doing. Secondly, users can monitor what their agents are doing by reading agent log files. Thirdly, users can read their agents' scripts when they return home.

In the first way, agents send their messages using HTTP post requests. Typically, agents inform their users every time they move within the agent network, come across a problem or perform a task. Wherever the agent is, it sends back to its user a message in the format shown earlier. The user's agent router checks the message type and its destination by parsing the message. It then re-directs the message to the user (wherever they are). The user's agent server receives the message from the agent router and updates the GUI(s) with the information. For example, a message sent by an agent to its user could be 'attempted to move to X and was refused'.

In the second way, users can read agent log files[10]. Wherever an agent is and whatever it is doing, it generates log entries that are written to log files. Every user has an agent log file on their computer. The log entries originate from two sources. Firstly, all the user's mobile agents send home log entries. Secondly, all visiting agents from other users generate local log entries. This enables users to track their own mobile agents and monitor what local agents from other users are doing on their computers.

In the third way, users can read their agent's script when they return home. This is made possible because agents can modify their own scripts. They can update their own scripts with information such as where they have been and what they did once they arrived. This reduces the need for

---

name abc.co.uk via web server aweb.co.uk, but the user's address is kyzyl@aweb.co.uk and not kyzyl@abc.co.uk.

10 Formats for agent log files will be described for the IAS in Chapter 6.

agents to keep sending messages home to their users and, consequently, reduces message traffic within the agent network. Though users cannot monitor agents in real-time this way, they will have a permanent record of their agent's actions in a concise form.

Every user within the agent network has a main GUI with three sub-windows, see figure 3.7. These windows are used to display textual information received by the user from their agents and visiting agents.

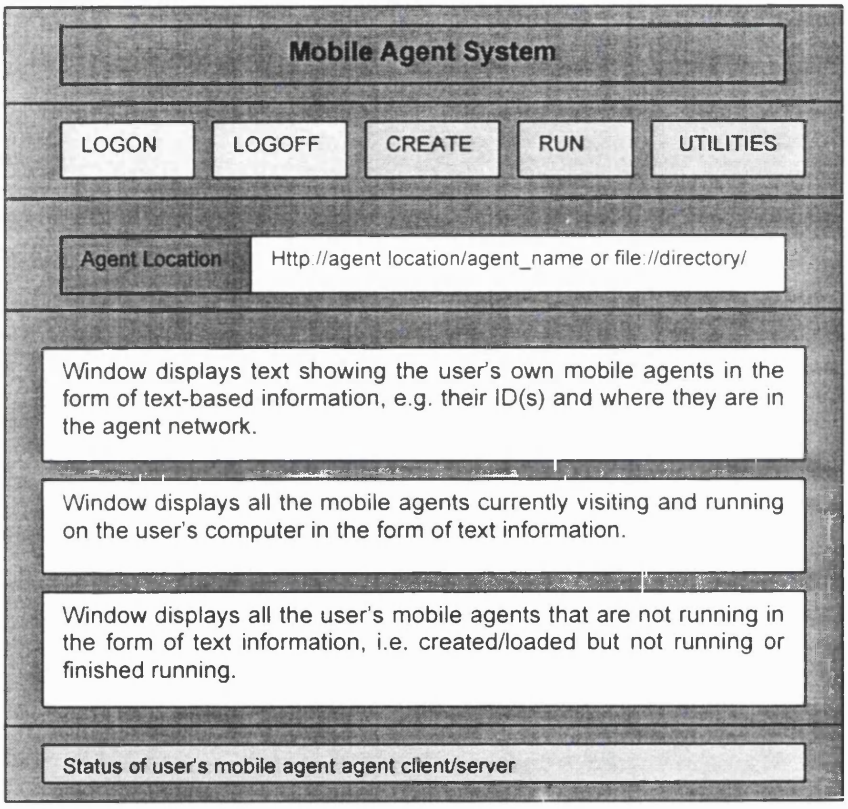


Figure 3.7 A simple GUI for a mobile agent system[11]

The first sub-window shows the user's mobile agents, i.e. which agent, where they are and their status etc. The second window shows agents that are visiting from other users. The third window shows non-running or completed agents. This agent pool can be used by the user to examine their scripts, edit them etc. with the various agent utilities. These agent utilities are described in the next section.

[11] See figure 5.4 for the screen dump of the Internet Auction System GUI.

Agent Utilities

Agent utilities enable users to save, load, edit, and parse their agent scripts. Users may load agents either from a local directory or from a remote web server. In the first case, the 'file' Uniform Resource Identifier (URI) scheme is used. In the second case, the *http* URI scheme is used. For example, to load a local agent script into their system, a user enters *file://c:/local_directory/agent_name*[12] in their GUI window. To download a remote agent, a user enters *http://web-server-address/agent_directory/agent_name* in the GUI window. However, users may save agents to local files but not to remote computers, though this could be implemented. Once an agent has been loaded or created dynamically, a user can view, edit, and if necessary parse the agent script. However, scripts created by the system do not need parsing.

In addition, the agent system has two special features, namely auto-loading and auto-saving. When users log into the system, they may opt for all agents in a pre-defined local directory to be auto-loaded and then run. In addition, at certain time intervals (e.g. when users log out), all resident agents on their system could be saved to a predefined local directory. This mechanism creates a persistent record of the agents resident on a user's system at a particular time. This is useful for two reasons. Firstly, if the computer crashes a permanent record of all the agents exists. Secondly, it enables users to log off without losing other users' visiting agents.

3.3 Achievements

This chapter described the requirements and design solutions for issues relating to agent mobility and management. The main requirements relating to the mobile agent system were platform independence; utilising the Web; forming distributed, dynamic and transient agent networks, and supporting mobile users. This would enable users to create flexible mobile agent networks, i.e. users may join or leave it at any time from any computer in the network. Therefore, users do not need a fixed address; they can be mobile. In designing the agent system, Java and Perl were used as the implementation languages to make the system platform independent. The infrastructure was based on a client/server architecture. It used the HTTP protocol and socket connections to network the agent client/servers. In particular, the HTTP post request was used to move agents from one computer to another. Each user would run an agent client/server on their computer enabling them to join the agent network. In addition, every user would be registered with a web server. This had three main benefits. Firstly, it utilised the Web as a backbone to the agent network. Secondly, it enabled users to join the agent network from any computer. Thirdly, the web server's security features could be used in the agent network. Once users had logged in, their

12 Note that in UNIX the path separators are '/' but in Windows they are '\'.

current location would the sent to their web server. If users wanted to send their agents to another user, they only needed to know the user's web server address and username. The agent would then be routed to the user's current location via the destination's web server.

The main requirements relating to agent management were: to enable users to create agents; to enable agents to autonomously explore their agent network; to enable users to monitor their agents; and to provide various agent utilities. In particular, users should be able to create agents that move freely in the agent network, exploring if necessary and returning home by a certain date. As the agents moved around the agent network, they used HTTP post requests to send back information to their users informing them of where they currently were and what they were doing. In addition, agents could modify their own scripts (see next chapter) so that they could concisely store information on where they had been and any problems they encountered. On returning, users could use various GUI(s) to view, edit and save their agents on web servers or in local file directories.

Therefore, in this chapter, a mobile agent system using agent client/servers and web servers was designed. This system enabled users to create, monitor, and utilise their mobile agents. These agents could freely move around the agent network forming a dynamic, transient network of distributed mobile users. The next chapter concerns how these mobile agents could be used to automate tasks at users' computers within the agent network.

Chapter 4 – Creating Autonomous and Adaptable Agents

This chapter continues and completes the discussion of the Adaptable Mobile Agent (AMA) by focusing on the tools and concepts associated with agent automation and adaptation. The first section describes how mobile agents can perform tasks and communicate with each other in a distributed system. The second section describes how users can adapt their agents to perform tasks well in an agent environment simulator. These are described in terms of requirements and design solutions. The last section summarises the achievements made.

4.1 Automating Agents

In the last chapter, agents could move around a dynamic agent network. Users could create mobile agents and schedule them to move from user to user, explore, and return by a specified date and time. However, these agents did not do anything useful as they traversed the network. Therefore, this section looks at how agents could be used to automate tasks as they move from user to user.

4.1.1 Requirements

The main requirements considered important for agent-based systems are listed below [SOM96]. However, some requirements overlap with previous requirements relating to agent mobility and management. In particular, how agents are scripted using scripting languages is more appropriately described here.

- Programmable Agents
- Agent Verification
- Concurrent Agents
- Persistent Agents
- Reusable Agents
- Stable/Efficient System

Programmable Agents

As mentioned in the last chapter, this thesis is not concerned with agent interfaces, although agent interfaces are becoming more sophisticated [BIC98], [CHO96]. However, users should still

be able to create agents using simple Application Programming Interfaces (API) and/or simple text editors. To make it easier for users to script their own agents, agents should be easy to read, write (script), and edit. In addition, an agent-based system should make it easy for users to reuse their agents.

Agent Verification

Once a user has created their agent script by inputting the various details via GUI(s). The system should automatically generate the agent script so that it successfully parses (lexically and syntactically). Moreover, users should be prevented from running their agents until the agent scripts have been successfully parsed and if possible checked for potential semantic errors e.g. type-checking. Once users have created their agents, they should be confident that the agent will do what it has been programmed to do. However, verifying computer programs is difficult [HAE87]. Therefore, it would be desirable to keep an agent's behaviour as simple as possible. This would make designing and checking an agent's behaviour easier.

Concurrent Agents

In a multi-agent system, users should be able to create and run as many agents as they wish. In addition, agents should be able to communicate concurrently with their users and with each other. However, there are three ways of constructing multi-agent system communications [CHO96]. Firstly, agents could be static at their respective computers. They could talk to each other by broadcasting or via dedicated lines of communication. Secondly, agents could be mobile, move around, and meet each other at a particular location (computer) to communicate. Thirdly, agents could be mobile and capable of communicating with remote agents. The first way has advantages and disadvantages. Firstly, agents that can broadcast messages or communicate directly with another are more secure; i.e. agents do not have to move to other computers and run. The second way (adopted in this thesis) would only allow agents to communicate with each other locally on a computer. The third way would combine both approaches and is generally more complex. For this reason, not many multi-agent systems have completely autonomous mobile agents that can remotely communicate with each other. The first and third way can present some problems in applications where information being distributed has to reach agents simultaneously. For example, in electronic auctions the distributed agents could receive asking bids at different times giving some agents an advantage and leaving others disadvantaged with outdated information [BIN97] and [HUB93]. The second way has advantages and disadvantages. One advantage would be that agents could be synchronised. Another advantage would be that agent communications would not take up network resources. A disadvantage would be that

agents have to meet at a computer to communicate with each other. They somehow have to know where and when to meet. In any case, checking what can and cannot happen in a multi-agent system would be very difficult, if not, impossible. Therefore, keeping agent communications simple and verifiable is important.

Persistent Agents

Often computer systems are criticised for being unreliable, and many users find it difficult to trust computers to do something on their behalf. User trust is an important issue because agents are more likely to be used in widespread electronic commerce systems [CHA96]. Therefore, to build up trust, agent-based systems must be reliable. However, external events can cause computers to fail or crash, e.g. power failure. Users would require their agent-based systems to be recoverable. In particular, agents and their states should be recoverable, i.e. persistent. This should enable users to restart their agent systems and rerun their agents from where they left off.

Reusable Agents

Users should be able to reuse their agents especially if agents are complex and take time to configure. They should only have to re-configure part of the agent. This is particularly relevant to adaptable agents because users may take a long time to adapt agents that perform satisfactorily.

Stable-Efficient System

As more users join the agent system, create agents and run them, the agent traffic within the system increases. This will undoubtedly affect the performance of the system. However, it is desirable that the system remains stable and does slow down to an unacceptable level. Therefore, it would be desirable to keep agent to user and inter-agent messaging to a minimum.

4.1.2 Design Solutions

This section describes how agents are used to automate tasks. The design solutions for agent automation fall into four areas. The first area concerns how agents are modelled using finite automata. The second area concerns how SGML is used to script agents. The third area concerns how agents are parsed and then interpreted as running threads. The fourth area concerns how concurrently running agent threads are controlled using agent monitors. The four areas are brought together in the section, Multi-Agent Systems. This section explains how all the design concepts could be used to construct an agent-based system.

Agent Automata

If agents are to automate processes and act on behalf of the user, they must be written or encoded in a computer language. However, before discussing how agents are modelled as finite automata, scripted and interpreted, their generic structure is described. Agents contain information and data. They also act and react to their environment. In this thesis, Adaptable Mobile Agents (AMA or agents in the rest of this thesis) consist of the following components, see figure 4.1.

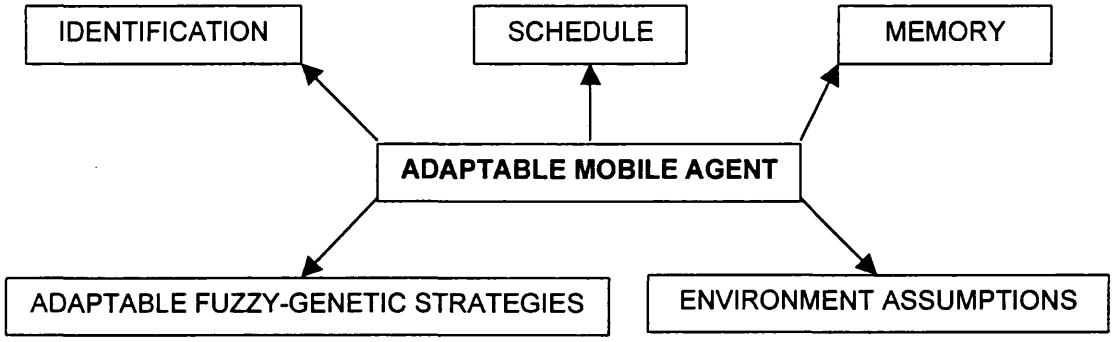


Figure 4.1 The Adaptable Mobile Agent generic structure

The IDENTIFICATION (ID) component uniquely identifies the agent from all other agents in the multi-agent system. The SCHEDULE component specifies a list of users that the agent will attempt to visit. The ENVIRONMENT ASSUMPTIONS component specifies an agent's assumptions about its environment. The ADAPTABLE FUZZY-GENETIC STRATEGIES component contains an agent's strategies for making decisions and performing actions in its environment. The MEMORY component contains an agent's current state. In addition, an agent can log its own activities as it traverses the agent network by updating its MEMORY component. This provides a permanent scripted account of the agent's activities. These components will be explained in detail for Seller and Bidder agents in Chapters 6 and 7.

An agent's generic components provide a static structure. However, an agent's behaviour has to be formalised so that they perform tasks and automate processes. One way of doing this is to formalise an agent's behaviour within its environment in terms of states, state transitions, actions, and events. On creation, an agent begins in a starting state and moves to a different state due to events in the agent's environment. This is called a state transition. For example, an event could be that it is time for the agent to return home. Every time an agent moves from one state to another, it could perform actions. These could be writing a log entry or moving from one user to the next. An agent keeps changing state until it reaches one of its terminating states. When it does this, the agent finishes and no longer runs as a process or thread.

To explain these ideas, the state transition diagram for a simple mobile agent is given in figure 4.2. In this example, a user creates an agent with a list of users to visit (observe) and a time to explore and return by. The agent decides where to go in the agent network according to its schedule. Once it has visited all the users within its schedule, it asks the local system for additional user addresses. It can now start exploring its agent network. Eventually, after it exhausts visiting and exploring, it returns home. The state transition diagram shows the mobile agent beginning in the STARTING state. As it moves around the agent network, it changes state (shown by arrows). Finally, when it returns home, its state changes to the HOME state. However, modelling agents as finite automata and defining their behaviour in terms of state transition diagrams becomes clearer when useful agents like Bidder and Seller agents are modelled. State transition diagrams for Seller and Bidder agents are described in detail in Sections 6.2.3 and 7.2.3. In any case, modelling agents as finite automata has two advantages. Firstly, they enable an agent's behaviour to be defined in terms of state transitions. Secondly, they enable agents to be saved in their current states and recovered.

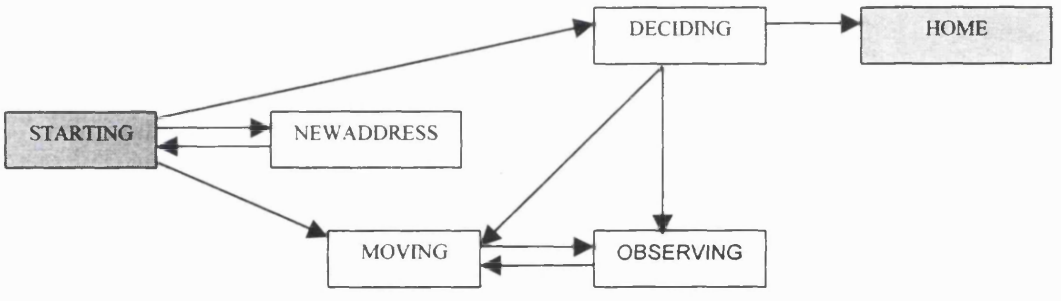


Figure 4.2 Mobile agent state transition diagram

Once an agent's behaviour within its environment has been modelled as a finite automaton (its state transition diagram, possible events causing these state transitions and its actions have been formalised), it needs to be implemented. In this thesis, three stages are used to do this. Firstly, an SGML-based mark-up language is constructed so that an agent's components can be scripted. Secondly, agent script parsers and interpreters are written so that the agent script can be parsed and interpreted. The interpreter extracts information from the script and runs the agent as a single thread of execution in accordance with its state transitions. Thirdly, agent monitors are written to control the concurrently running agent threads. The following few sections describe these stages in turn.

Agent Scripts

In this thesis, agents are implemented as mobile scripts, other techniques can be found in [BI96], [BRE98], [HUR97], and [KEN98]. Their identification, intelligence, assumptions, security etc. are scripted using specially tailored SGML-based script languages. However, there are hundreds of agent languages [FIN], [WOO95], so why choose SGML to create new languages? The main reason is to introduce the concept of using SGML to tailor agent scripts to the application. More specifically, SGML-based scripts were chosen to implement agents for the following reasons. Firstly, agents constructed using SGML languages tie together agent and web technologies, highlighting the possible analogies between web pages, intelligent mobile documents and agents. Secondly, SGML is a standard for creating other mark-up languages and structuring documented information. With the growth of hypertext, media, and virtual reality applications using mark-up languages [HYT], [BOS97], [MAT98], and [XML], the next step in the development of mark-up languages would seem natural to be in the agent application area. Therefore, these agents would become a standard within a particular application area e.g. Internet auctions. Thirdly, the scripts could be self-explanatory. Users writing the scripts can see exactly what their agent represents, i.e. what information it contains. Fourthly, agent scripts could be verified and enforced using Document Type Definitions (DTD) for the agent script language; again analogising agents with web pages that are parsed using the HTML DTD. A DTD is a parser generated by SGML management systems that check documents are structured correctly. SGML would make this task easier, providing a common platform with which to define agent scripts. Moreover, the natural extension to downloading HTML pages and viewing them in a web browser would be to have mobile SGML-based agents download to 'agent-browsers' so that users may view them graphically. Therefore, SGML can be used to create simple mark-up languages [POP96], [SMI92], [TUR96]. These mark-up languages could be used to mark-up and structure agents. For example, the agents discussed so far consist of the following components, IDENTIFICATION (ID), ENVIRONMENT ASSUMPTIONS, SCHEDULE, FUZZY-GENETIC STRATEGIES and MEMORY. Within each component, an agent's structure can be broken down further. For example, the ID component could consist of the name of the user that created it, their address, and the date and time it was created. Therefore, an agent's structure can consists of components; sub-components and so on until it cannot be broken down further. In some cases, a component may have more sub-components of the same type; e.g., a SCHEDULE component may contain any number VISIT sub-components. A VISIT sub-component could contain details such as the user's address and what to look for at the user's destination etc.

The three main methods used in SGML to structure (mark-up or tag) information are *elements*, *attributes,* and *entities*. *Elements* would be an agent's logical components that make up its

structure, e.g. SCHEDULE or sub-component VISIT. *Attributes* could contain meta-information about an element. *Entities* would be units of information that are marked-up, e.g. a string or an integer. If an agent's structure is completely known then an SGML DTD can be written. An SGML DTD would define all the possible scripts for a particular type of agent, see figure 4.3. It would specify the mark-up (or tagging rules of all agents of a particular type) by formalising what tags are used to mark the agent up, the number, order etc. However, agents may have different numbers of sub-components or contain different information but they must follow the tagging rules specified in the DTD.

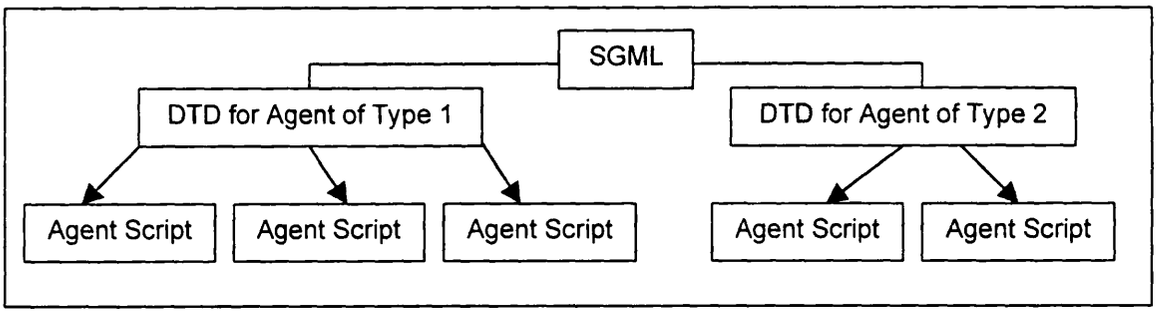


Figure 4.3 SGML as a meta-language

An example of an agent script is given in figure 4.4. This shows an agent script with its structure marked up using opening tags (tags that start with '<') and closing tags (tags that start with '</'). The agent script starts with a <AGENT> starting tag and ends with a </AGENT> closing tag. Every component and sub-component can be tagged with a starting tag and a closing tag, so on until the sub-components like, <CREATOR> or <DATE CREATED> which contains information that cannot be broken down further. In any case, a fully tagged agent script is ready to be parsed and interpreted using agent parsers and interpreters.

```
<AGENT>
    <ID>
        <CREATOR>kyzyl</CREATOR>
        <USER ADDRESS>Mongolia</USER ADDRESS>
        <DATE CREATED>12/12/98</DATE CREATED>
    </ID>

    <ENVIRONMENTAL ASSUMPTIONS>
    </ENVIRONMENTAL ASSUMPTIONS>

    <SCHEDULE>
        <VISIT></VISIT>
        <VISIT></VISIT>
    </SCHEDULE>

    <FUZZY GENETIC STRATEGIES>
    </FUZZY GENETIC STRATEGIES>

    <MEMORY>
    </MEMORY>
</AGENT>
```

Figure 4.4 An example of an agent script using tags

Agent Parsers and Interpreters

Agents have been described as scripts written in an SGML-based language that mark up the agent's components. In addition, the agent has been formalised as a finite automaton that consists of states, state transitions, actions, and events. The next stage is to take the agent script, parse it, and make it run as a finite automaton. Since the agent's script language is based on SGML, any agent script could be verified by its DTD to see if it follows the rules implicit in the DTD. These DTD parsers would check the script lexically, syntactically and semantically. Lexical checking makes sure the agent script has the correct number of tags. Syntax checking makes sure the agent script has tags and tagged data in the correct order. Semantic checking makes sure the agent script tags the correct type of information. If an agent script has been parsed successfully, it can be interpreted.

```
// Agent Script
// Agent script is passed as a string to the interpreter's constructor
// Extract data and allocate to variables
state = START

// Infinite loop
for(;;){

   if state = START {

            // Check Environmental Events
            // State Transitions
            If event = E1 then state = X
            else if event = E2 then state = Y
            // Actions e.g. change agent script's state and log information
   }
   else if state = X {

            // Check Environmental Events
            // State Transitions
            // Actions e.g. change agent script's state and log information
   }
   else if state = Y {

            // Check Environmental Events
            // State Transitions
            // Actions e.g. change agent script's state and log information
   }
   else if state = HOME{

            // Environmental Events
            // No State Transitions
            // Actions e.g. log information
            // Terminate agent interpreter thread
   }
}
```

Figure 4.5 Pseudo code for an agent interpreter

The agent interpreter extracts an agent's state and other information from the tagged script and uses it to automate whatever the agent is supposed to do (according to its state transitions and actions). An agent script is passed to the agent interpreter's (Java object's) constructor as a text string. This Java program has all the agent's states, state transitions, and actions already programmed into it (figure 4.5). Initially, the agent would start in its STARTING state. However, as events happen they affect the agent's state. In turn, an agent may perform actions whilst in a new state, e.g. updating its own state and logging information by modifying its own script. This continues until the agent reaches its terminating state. The interpreter thread is then terminated by the agent system and the agent script remains inactive.

Agent Monitors

This section aims to briefly describe the ideas of how agents would communicate with each other in a multi-agent system. Agent interpreters run as threads within the agent system. When agents want to communicate with their remote users, they use agent routers. However, the agent systems designed in this thesis only allow agents to communicate with each other if they are physically located on the same computer. This would reduce network message traffic and speed up agent communications. In addition, any information that has to be broadcast to many agents (e.g. asking price in an auction) would reach all the agents (in effect) simultaneously. If the agents were distributed on remote computers, these messages could reach some agents before others which could be unfair and difficult to rectify [BIN97], [FS98]. Agent monitors could also be written to control all the locally running agent threads. This could be used to impose synchronisation, avoid agent deadlock, busy-waiting, live-lock (taking all resources) and other problems associated with concurrency.

An agent monitor would consist of a program with various synchronised shared-variables and the methods required by agents to access these variables. For example, if an agent wanted to set variable X, it would use a write method to set X. However, the agent can only do this if some criteria are met, e.g. no other agent is currently reading the variable. By having shared-variables and methods that enforce agents to behave and interact in a synchronised way provides a very efficient and stable method for inter-agent communication. This is implemented in the IAS as an auctioneer monitor, see Chapter 5. Also, see Appendix C for auctioneer monitor's source code.

Multi-agent Systems

The main concepts discussed in agent automation were agent components; finite automata; SGML; agent parsers; interpreters; threads and monitors. An agent could be considered as being

made up of components that embodies or encodes its features. Its behaviour could be modelled as a finite automaton in terms of states, state transitions, actions, and events. SGML could be used to design script languages that would specifically mark-up an agent's various components and states. This would standardise the agent, making it portable for all users within the agent system. In addition, an SGML DTD could be used to verify the agent script lexically, syntactically and semantically. An agent interpreter could then be used to automate processes within the agent system by extracting the information from the script and running the agent as an automaton. The interpreter checks the agent script for its current state and performs actions on its behalf. As the environment changes (events occur), the interpreter changes the agent's state by modifying its script and performing its actions. During this process, agents may communicate with each other only if they are located on the same computer. However, they may communicate with their users at any time. This simplifies agent communication and reduces network message traffic making agent-based systems more stable and efficient. To control agent communication during inter-agent communications, an agent monitor is used. This consists of shared-variables and methods that the agents use to communicate with each other. This enables all agent communications to be synchronised. Eventually, an agent's state will change to a terminating state and the agent will stop running.

Together, these tools and concepts could be used to design multi-agent systems in which every user within the system has agent parsers, interpreters, monitors and GUI(s) to interface and interact with the system. Therefore, as agent scripts move from user to user, they could be interpreted and controlled locally, enabling distributed mobile agent systems to be constructed.

4.2 Adapting Agents

This section explores how some agent-based applications could be modelled as games in which the agents are players. The rules of the game could be used to construct a simulator to simulate the agents' environment. In addition, the information used by agents to play the game could be encoded using fuzzy logic, their strategies encoded using genetic structures, and Fuzzy-Genetic Algorithms (FGA) could be used to evolve them. Users would then use an FGA-based simulator to test and adapt their agents to perform their strategic tasks better.

4.2.1 Requirements

The following list specifies two groups of requirements. The first group concerns creating a simulated agent world, i.e. simulator. The second group concerns the algorithms used by the artificial agents to improve their performances during successive simulations.

- Environmental Simulator
  - Configurable Assumptions
  - Multi-agent Environments
- Learning Algorithm
  - Robust and Flexible
  - Configurable Adaptation Time
  - Embedded Strategies

The Environmental Simulator

Simulators should enable users to test their agents before using them in real applications. To configure an artificial agent, environment parameters and assumptions have to be set. It is difficult to generalise which parameters are necessary for a simulator because this depends on the application. In this thesis, only relatively simple artificial auction environments are modelled. In any case, there are three main reasons why simulators may be beneficial to agent-based systems. Firstly, they give users some confidence that their agents would perform their tasks well. Secondly, users can configure simulators to test out their agents in different environments in real-time. Thirdly, if an agent's makes strategic decisions (tasks)[1] then they could be adapted within the simulator using a learning algorithm.

The Learning Algorithm

To explain an algorithm's robustness assume that the agent's environment can be modelled mathematically in the form of a time dependent equation. The equation's inputs are an agent's decision (say a number), other agents' decisions, and environmental assumptions (other numbers). The equation's output is the outcome. If the best decision an agent can make is the one that results in the largest output (for the particular input assumptions) then the agent's decision is optimal. The problem is that an agent's environment and decisions are often difficult to mathematically model [HAE87]. Moreover, even if they are not, it is often difficult to find optimal solutions, e.g. NP-Complete problems [GRE89]. Robust algorithms have a characteristic that enables them to overcome some of the inherent difficulty in finding good solutions to problems [GOL89]. They do not guarantee to solve the problem with the optimal solution but they usually find near optimal solutions. Algorithms that find near optimal solutions quickly maybe more suitable for practical agent systems than algorithms that find optimal solutions very

1 An agent's tasks could be a strategic decision, e.g. what to bid in an auction? Alternatively, it could be something non-strategic, e.g. actually submitting a bid. In this thesis, only strategic tasks or decisions are evolved.

slowly. Besides robustness, an algorithm should be flexible. It should be capable of modelling a variety of environments. Moreover, it should be configurable so users have some control over how long the algorithm would take to evolve good strategies. Finally, strategies should be encoded and embeddable within an agent's script allowing the agent interpreter to convert the agent's encoded strategies into decisions (Boolean or numeric) as and when necessary. The whole process of configuring the environment, running the algorithm, selecting the strategies and embedding them ready for the agent to use, should be seamless. If a learning algorithm with the above qualities is incorporated into the simulator then users should have full control on the quality and quantity of their agent's strategies.

4.2.2 Design Solutions

Designing general-purpose agent simulators and learning algorithms is beyond the scope of this thesis. Therefore, only concepts will be discussed here. However, a working auction simulator and its learning algorithm has been designed and built for the IAS (see Section 5.2). In any case, three questions need to be answered in respect of implementing and analysing simulated agent environments and learning algorithms.

- How is an agent environment modelled?
- How are an agent's decisions modelled?
- How can an agent's strategies be improved?

The Simulated Environment

If an agent's environment can be described as a game then this would provide a useful framework for implementing a simulator and its learning algorithm. The main elements of a game are players, rules, information, strategies and payoffs [BES89], [BIN92], [RAS96]. Therefore, the following analogies can be made:

- The *players* are the agents;
- The *game* (rules) is the agents' simulated environment or system;
- A player's *information* corresponds to an agent's knowledge base;
- A player's *strategies* are the strategic decisions an agent makes within its environment;
- *Payoffs* are the rewards or losses that an agent incurs within its environment.

A simulated environment (game) determines what agents can and cannot do and when they can do it; they define the rules. Though most games are competitive, some are co-operative [RAS96].

Only competitive environments are considered because agent-based commerce systems are more likely to involve competitive agents, e.g. simulated auction markets. As with most competitive games, agents attempt to win by using strategies. Their strategies may use prior information (beliefs) and posterior information[2] gained during the game. The set of strategies that an agent could use to make a decision at a particular point in a game is called the strategy space. Finding good strategies can be very difficult especially if the strategy space is very large. However, if an agent is going to improve its chances of winning, it requires good, or, if possible, optimal strategies, i.e. strategies that maximise expected payoff. Therefore, an agent must have good strategies for every decision it has to make in every possible state of the game. This creates three problems. Firstly, the possible states of a game can be very large. Secondly, the set of strategies for at a particular state in the game can also be large. Thirdly, agents need to carry around with them a vast set of strategies to have a chance of performing well. The next section attempts overcome these problems by modelling the environment using Fuzzy-Genetic Algorithms (FGA). These are used to encode a game's state space, and an agent's information and strategy spaces.

The Fuzzy-Genetic Algorithm

Once a game's rules have been defined, the players (agents) make decisions and attempt to beat their opponents. However, to model a game, five issues are considered. Firstly, what information is at hand for the agent to use? Secondly, what strategies are available to an agent at every point in a game? Thirdly, how can a game's state space and agents' information and strategy spaces be partitioned, or bounded, so that they do not need an unreasonably large set of strategies? Fourthly, how do agents utilise this information and make a strategic decision? Fifthly, how can agents learn by playing repeated games? These are discussed in the next few sections.

Information and Strategies

In game theory, players use information to make strategic decisions. It can be categorised into perfect, certain, symmetric and complete- and their converses. These define what information is available to the agents in a particular game. Symmetric information implies all agents have access to the same information, whilst asymmetric implies inequality, i.e. private information. Complete information assumes agents have access to all the available information, whilst incomplete information implies some information is not available. For further details see [RAS96]. However, it is important to recognise what pieces of information are important or relevant to the agents, and whether they all have equal access to this information. This is important because each agent only uses this information to make its strategic decisions.

2 An agent's information can be private of public depending on the type of game.

Therefore, to simulate an environment, firstly, the rules of the game are established. In the case of a complex environment, assumptions would be made to make the modelling easier. Secondly, the most important pieces of information used by agents to make decisions need to be formalised. Thirdly, the simulator should be configurable so agents can have equal access to information. Lastly, agents' decisions should be formalised in terms of the information used and range of possible strategies that can be made[3].

Mapping State Spaces

During a game, agents may make a number of strategic decisions. These strategies represent rules that agents follow at every state of the game using their information. The larger the state space, information space and number of decisions, the greater the size of the strategy space. Because agent strategy spaces can be very large, fuzzy sets [KOK92], [ZAD74] and genetic structures [HOL92], [GOL89] are used to partition these state spaces. Fuzzy sets are used to partition the state of the game for four other reasons. Firstly, it may not be practical for mobile agents to carry large sets of strategies around with them. Secondly, it makes the task of finding good strategies easier because there are fewer. Thirdly, it is flexible i.e. the partition can be made coarser or finer[4]. Fourthly, more emphasis can be given to important parts of the state space.

Exactly how the fuzzy sets partition information and game state spaces depend on what pieces of information are included. For example, assume ordinary sets X, Y and Z completely specify an agent's information space and the game's state space. Therefore, at a particular state of the game an agent only uses three elements *x, y, z* from the sets X, Y and Z to make its decision. However, an agent requires a strategy for every state. If the number of elements in X, Y and Z are |X|, |Y| and |Z| respectively then agents may need a very large number of strategies for the game (= $|X|*|Y|*|Z|$). If fuzzy sets are used to partition X, Y, Z in a meaningful way, i.e. use more fuzzy sets for states that occur more often, and fewer otherwise, then the number of states can be reduced dramatically. For example, if *n1* fuzzy sets cover X, *n2* cover Y and *n3* cover Z then the number of agent strategies reduces from ($|X|*|Y|*|Z|$) to (*n1*n2*n3)* strategies. Although, the number of strategies reduces, they may be less effective. In any case, there are many ways to implement fuzzy sets; in this thesis, the mapping is simple. In figure 4.6, five fuzzy sets are used to partition X where the elements of X are all positive integers. However, for the purpose of simulation, only integer elements in [A, B] are important, where A and B are positive integers

---

3 In Chapter 5, these ideas are implemented to design an auction simulator. The auction environment is represented as an auction game. The players are the Bidder and Seller agents. The auction method defines an auction's rules and its type defines the type of information the agents use, e.g. symmetric incomplete private values. See Section 1.2 where auction methods and types are explained.

4 This has interesting implications to competing agents. Do agents that use finer state spaces always do better than agents that use coarser state spaces? This is investigated in Section 8.2.

and A < B. Hence, only one fuzzy set covers all integer elements < A and one all high integer elements > B.

A membership value quantifies how much an integer element belongs to a fuzzy set. These membership values lie in the interval [0,1], such that for a particular fuzzy set, an element with a membership value of one implies the element fully belongs to the fuzzy set and a membership value of zero implies the element does not belong to the fuzzy set.

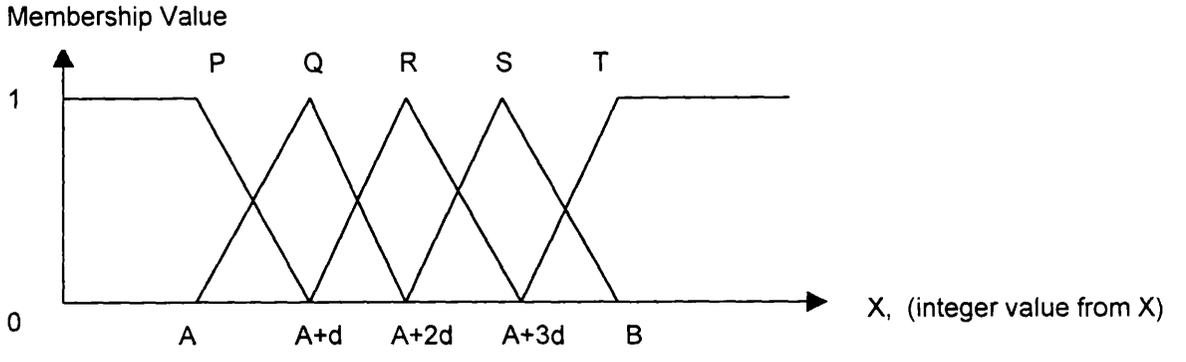


Figure 4.6 Fuzzy set partitioning

Partitioning state spaces should be rational but can be subjective. Figure 4.6 shows how fuzzy sets are used to partition all state spaces in a special way. For example, if Q and R are overlapping fuzzy sets with membership functions,

$$\mu 1 and \mu 2 then \qquad \mu 1(x) + \mu 2(x) = 1 \forall x \in Q \cap R$$

If the sets overlap in this particular way then formulae for the membership values can be derived for all fuzzy sets and all integer elements x in set X. See Appendix A for derivation of membership value formulae. These formulae are adopted for all fuzzy set partitioning, see Sections 6.2.1 and 7.2.1 for examples. Similarly, the information sets Y and Z can partitioned using fuzzy sets. However, different numbers of fuzzy sets could be used to partition them (e.g. three fuzzy sets cover Y and two fuzzy sets cover Z) enabling the designer to focus on important areas of the state space[5]. Therefore, for every state of the game (including an agent's information states), described by the triplet *(x, y, z)*:

- Five membership values for x correspond to the five fuzzy sets that partition X;
- Three membership values for *y* correspond to the three fuzzy sets that partition Y;
- Two membership values for *z* correspond to the two fuzzy sets that partition Z.

5 Membership functions for fuzzy sets that cover X, Y, and Z are likely to be different but the fuzzy sets still obey the overlap rule.

These membership values are used to summarise the state of a game and all the information used by agents to make their strategic decisions. The next section, Encoding Strategies describes how genetic structures are used to encode agent strategies. The section after that, Strategic Decisions, describes an example of how membership values are used to summarise information used by agents and the state of a game. In addition, it describes how agents use these membership values together with their encoded strategies to make strategic decisions.

Encoding Strategies

During a game, agents are required to make strategic decisions. In some games, agents may only make one decision, in others they may make many, possibly in some particular order. This is called the order of play. In any case, every strategic decision is encoded using genetic structures. Genetic Algorithms (GA) are used to encode and evolve agents' strategies for three reasons. Firstly, GA(s) adopt effective adaptation processes that mimic natural selection. Secondly, they have a reputation of being robust, i.e. can find good (not necessarily optimal solutions) in complex strategy spaces. Thirdly, they have a reputation of being flexible; i.e. they can be applied to many optimisation problems. The agents' strategy spaces are encoded using a variety of genetic structures, as follows (see [GOL89] for general definitions):

- An agent's *genotype* encodes all its strategies, for all its decisions, information states and game states;
- An agent's *chromosome* encodes all its strategies for a particular decision;
- An agent's *gene* encodes one strategy for one particular information/state of the game;
- A *gene's locus* (position on its chromosome) corresponds to an agent's strategy for one particular information/game state;
- *Gene alleles* represent a gene's permitted values or codes, e.g. integer values from the set {0,1,2,3,4,5,6,7,8,9};
- The *phenotype* determines the agent's strategy for a particular information/game state.

The following example is used to explain the genetic encoding. Assume three sets (X, Y and Z) describe the information used by an agent to specify the state of a game. Also, assume X is partitioned with four fuzzy sets, Y is partitioned by three fuzzy sets, and Z is partitioned by two fuzzy sets. So altogether, there are *(4*3*2 = 24)* possible fuzzy state combinations that could specify the state of the game. Assume an agent uses the fuzzy states to make a decision. Since, the chromosome encodes all these strategies; there must be a mapping from each fuzzy state to a strategy. The mapping is as follows. A chromosome consists of n genes, where n is the number of possible fuzzy states for a particular decision. Therefore, in this example, since there are 24

fuzzy states, 24 genes make up an agent's chromosome. Other decisions may have more or fewer genes depending on what sets are required and how many fuzzy sets are used to partition them. Two further properties are required to fully specify an agent's chromosome. Firstly, a gene's locus, i.e. its position in the chromosome. A locus of a gene relates to a particular fuzzy state of the game.

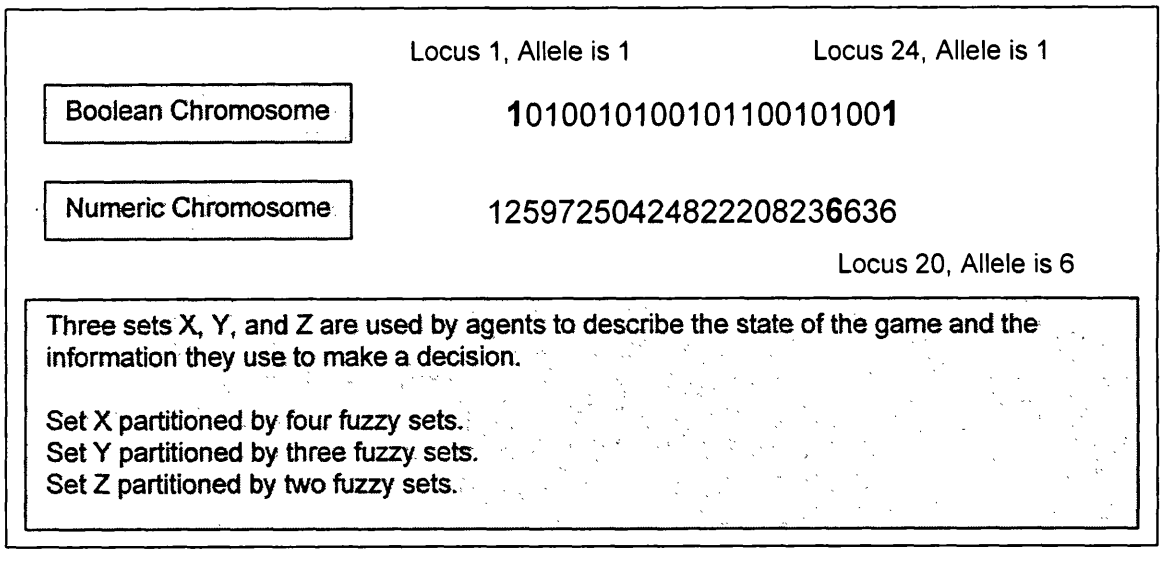


Figure 4.7 Boolean and numeric decision chromosomes

For example in figure 4.7, the first gene (locus 1) on the agent's decision chromosome only has phenotypic effect when the information and game states can be described (partially or not) by the fuzzy state. The next gene (locus 2) relates to the next fuzzy state and so on until the 24^{th} gene. A gene's expression or phenotype depends on what type of decision the agent can make. If it is a Boolean (yes or no) decision then gene alleles could be represented as either 0 or 1. If an agent makes a decision that requires deciding a numeric quantity (e.g. how much to pay for something?) then gene alleles could be bounded positive integers. Therefore, when the state of a game relates to a particular locus on the agent's decision chromosome, it will use the gene allele (Boolean or integer value) to make its decision. The next section describes how agents make decisions given the fuzzy membership values and their chromosomes.

Strategic Decisions

To see how an agent would make a decision, the numeric chromosome given in the previous example is used. Let an agent's decision chromosome be stored in an array of 24 elements (called DC[24]). Each element could be an integer between zero and nine depending on the agent's particular strategy. The information and game state just before the agent makes its decision is summarised as the triplet *(x, y, z)* where *x* is the value from set X, *y* is the value from set Y and *z*

is the value from set Z. Then x is stored as an array of four elements, where element 1 is the membership level of x in the first fuzzy set. Element 2 is the membership level of x in the second fuzzy set and so on for the other two elements. Two other arrays are formed for values of y and z. If F1[4] is the array of membership values for the value x, F2[3] is the array of membership values for the value y and F3[2] is the array of membership values for the value z then the 24-element array (called FS[24]) is formed as follows:

$$FS[k + 2*(j-1) + 6*(i-1)] = F1[i]*F2[j]*F3[k] \text{ for } i=1,2,3,4;\ j=1,2,3 \text{ and } k=1,2.$$

So each element of FS[24] corresponds to a unique fuzzy set combination of (X, Y, Z) and its value is the product of the three membership values. The agent reaches a decision by using the FS[24] array and its decision chromosome DC[24] as follows: firstly, FS [24] and DC [24] are multiplied element by element. The elements are added together, giving a value between 0 and 9. This value can then be decoded to give a number that represents the agent's decision. Examples will be given in Chapters 6 and 7 in which Seller and Bidder agents use their numeric decision chromosomes to decide how much to sell and bid in auctions. The same method is used for all non-Boolean decisions. If a Boolean decision is required, the arrays are multiplied and summed as before. However, the value will be between zero and one. If value is <0.5 then it is a 'no' decision otherwise it is a 'yes' decision.

In summary, agents may use various information sets at various stages in the game to make Boolean or numeric decisions. Fuzzy sets are used to partition the information and games state spaces to form a fuzzy state space that has fewer states. Agents have one decision chromosome that encodes their strategies for each decision. Together, these decision chromosomes make up an agent's genotype. Each gene corresponds to a particular fuzzy state that is only expressed when the values representing the agent's information and game states have membership products > 0.

Evolving Agent Strategies

For some games, the number of possible strategies required by players can be very large. Even after using fuzzy sets to partition information and game state spaces to reduce the number of possible strategies, the strategy space can still be large. For example, a chromosome that has 24 genes and each gene can take an integer value from zero to nine has 10 to the power of 24 or 1,000,000,000,000,000,000,000,000 possible strategies. This makes finding good strategies difficult for algorithms and theorists alike, [EPS74], and [GRE89]. Therefore, to make modelling simpler and the task of finding good strategies easier, assumptions are made. These can be implicitly used within a game's rules (e.g. auction types) or configurable by the user by setting

parameters, e.g. number of generations used to adapt agent strategies. For the agent environment simulator and Fuzzy-Genetic Algorithm (FGA), parameters fall into three groups, see table 4.1.

Environment Parameters	Parameters specific to the environment For example: Agent Strategy Type (see below) or in an auction, agent's initial credit.
Fuzzy Set Partition Parameters	Number of fuzzy sets used to *partition* information X Number of fuzzy sets used to *partition* information Y Number of fuzzy sets used to *partition* information Z ...
Genetic Algorithm (GA) Parameters	Total Number of *Generations* *Population size* in each generation Total *number of games* played by each agent *Crossover Probability* *Mutation Probability* *Gene Alleles* in Chromosome 1 *Gene Alleles* in Chromosome 2 *Gene Alleles* in Chromosome 3 ...

Table 4.1 Agent environment simulator and Fuzzy-Genetic Algorithm parameters

The 'GA' parameters are all standard except for the gene alleles; these encode an agent's decision for a particular fuzzy state. The 'Fuzzy Set Partition' parameters define how many fuzzy sets are used to partition information and game state spaces. The 'Environment' parameters are specific to the agent environment e.g. agent strategy type that sets which agents have fixed strategies and which have evolving strategies. In some games, there may be more then one type of agent. For example in an auction, there are two types, bidder, and seller. An agent could have either fixed strategies or evolving strategies, e.g. one bidder evolves, all other bidders and seller have fixed strategies, or all bidders have fixed strategies and seller evolves. These parameters are used to configure the FGA and simulator to create an artificial environment for the agents to evolve in. The next section describes the various stages that this FGA uses to evolve agent strategies in two parts. Firstly, how the first and subsequent agent populations are generated. Secondly, how agents' chromosomes are combined to form a new population.

Agent Generations and Forming New Populations

Once all the simulator's parameters have been set, the FGA can be run. The first generation is formed as follows: each agent in the population is taken in turn to be given a randomly generated genotype. Therefore, all agents start with random strategies. For all the remaining generations, agents do the following: each agent attempts to play a round of games. At the end of an agent

round, its performance or fitness is calculated according to some criteria, e.g. profit or loss. Agents are then ranked according to their performance. Agents are paired off with the agent ranked first with the one ranked second; the one ranked third with the one ranked fourth until all the agents have been paired (even population assumed). To maintain a stationary population and impose the rule that fitter agents have more children, the following formula is used. If p is the pth best pair, then $C(p) = -8p/(N-2) + 4N/(N-2)$ *rounded to the nearest integer* gives the number of children, where N is the population size. See Appendix A, for the derivation and explanation of the assumptions used in formulating this equation.

Each agent child is formed by its agent parents crossing their chromosomes and through mutating genes. There are 'GA' parameters to set the crossover and mutation probabilities. The child's chromosomes are formed one at a time and independently. If crossover occurs, the parent who supplies the first part of the chromosome and the parent who supplies the remainder are decided randomly. If crossover does not occur and only one parent supplies the whole chromosome then this is also decided randomly. The mutation probability determines the chance that a child's gene will mutate. For example, if a gene can take any integer value between zero and N, and is currently x then it may mutate with equal probability to x-1 or x+1. Except if x is already 0 or N then it can only mutate to 1 or N-1 respectively. For Boolean gene alleles, 0 would mutate to 1 and vice versa. The mutation process is applied to every gene in the agent child's genotype. Once all the agent children have been generated, they form the new generation.

Assessing an Agent Population's Performance

The FGA output summarises each generation's performance. Once all agents of a particular generation have finished, the performances for the best, average and worst agents are calculated and displayed. This shows if successive generations are improving or not. After all the generations have been generated, the performance of each agent from the last generation is displayed. This shows in detail how well each agent performed in that particular simulation. Users may choose their agent strategies by selecting the fittest agents from the last generation and embedding their strategies into real agents. These are then ready to be released into the real agent application e.g. real Internet auctions.

4.3 Achievements

This chapter completed the description of the AMA. In particular, autonomous agents should be easily programmable, verifiable, run concurrently, be persistent, be reusable and form an efficient and stable system. These requirements are implemented as follows: SGML could be

used to design agent script languages. These languages would mark up an agent's components (identification information, states, and strategies) and formalise its structure. Developing SGML-based script languages would make verifying agent scripts easier especially if SGML DTD(s) were constructed. An agent's behaviour could be formalised in terms of states, state transitions, events and actions it performs within its environment, i.e. as a finite automaton. In addition, agent interpreters and monitors would be required at every computer in the agent network. This would enable agents to run and automate processes locally. Moreover, agent monitors would mediate all inter-agent communications by enforcing synchronisation.

Adaptable agents require a configurable multi-agent environmental simulator and a learning algorithm to test and evolve their strategies. Designing a simulator would be made easier if the multi-agent environment could be modelled as a game. The agents (players) attempt to win using information from the game together with their strategies. In this thesis, fuzzy logic and FGA(s) are used to encode and evolve agent strategies. Both methods are regarded as robust and flexible for optimisation problems [GOL89], [ZAH74]. Fuzzy sets could be used to partition an agent's information and game state spaces. This would reduce the number of states used to specify the game; consequently, agents would require fewer strategies. This would reduce the search space for the FGA and thus shorten the time taken to adapt reasonable strategies. On the down side, it leaves the agent with imprecise strategies. Genetic structures could be used to encode agent strategies. In particular, a chromosome would represent all the strategies for one particular decision and each gene would correspond to a single strategy for a particular fuzzy state of the game. The FGA could then be used to evolve agents' fuzzy-genetic strategies by allowing the agents within a population to play a round of games. The agents that win the most games and receive the highest payoffs are deemed the fittest agents within the environment. The fittest agents form a new generation by ranking the agents, pairing the best agents with each other and by having more children using crossover and mutation operators. The new generation, consisting of the agents' children (parents are eliminated), plays a round of games as before. This process continues for the number of generations. The last generation should consist of fitter agents with better strategies than previous generations. Users may then choose which strategies they want their agents to use in real agent applications.

The tools and concepts discussed in this and the previous chapter give an idea of how a generic agent system could be modelled. (Although, enforcing agent-based solutions on computer systems may not always be the best approach see [WOO98a]). However, to demonstrate these ideas, auctions are modelled as games in which Bidder and Seller agents are the players competing in the simulated auctions. The next three chapters use most of the tools and concepts described in the last two chapters to build the novel IAS.

Chapter 5 - The Internet Auction System

This chapter describes a novel Internet Auction System (IAS). [However, Chapter 6 fully describes the Seller agent and Chapter 7 fully describes the Bidder agent]. Though, the IAS infrastructure and auction simulator are part of the IAS, for clarity, they are described separately. The first two sections describe the IAS infrastructure requirements and design solutions. Then the following two sections describe the auction simulator requirements and design solutions. Since, the IAS design utilises the tools and concepts explored in Chapters 3 and 4, these are briefly discussed in the context of automating auctions. The chapter ends with the achievements made by designing and implementing the IAS.

5.1 Infrastructure

The are two reasons for developing the IAS. Firstly, to enable users to automate auctions on their computers using Seller agents. Secondly, to enable users to find these auctions over the Internet and bid in them using Bidder agents. Therefore, the IAS relies on two types of agent to automate the selling and buying in these computer auctions. The Seller agent is adaptable but not mobile, i.e. it remains at its user's computer. The Bidder agent is adaptable and mobile, since it is required to move around the Internet and find the remote auctions held by Seller agents. However, before the IAS is described in terms of agents and their technologies, the general requirements for an IAS are described. In particular, the IAS requirements are explained in terms of what users (as sellers or bidders) may require from an auction system.

5.1.1 Requirements

The requirements that apply to all users whether using the IAS to sell or buy items are described first. Next, the requirements for a user (seller) who wants to auction something from their computer are described. Finally, the requirements for a user (bidder) who wants to bid for something in remote auctions are described.

- Firstly, the IAS should enable users (as sellers) to hold one or more fully automated auctions on their computers. In addition, it should enable users (as bidders) to find remote auctions and bid in them[1]. Therefore, the IAS should enable users to use their computers to

[1] Money transactions at the auctions are ignored.

simultaneously buy and sell items over the Internet. Each user (acting as a seller and/or bidder) within the IAS uses their computer as an *auction site*. Secondly, users should be able to independently form a distributed, dynamic and transient networked auction system. They should be able to join, leave, and rejoin by logging into the auction system, running auctions at any time and logging out of the auction system. However, they should not log out of the auction system when their auctions are running on their computer, e.g. bidding in auctions, but may do so otherwise.

- Secondly, users should be able to join the auction system from any computer on the Internet irrespective of its operating system; i.e. the auction system should be an open/platform independent system.

Automating Selling

The IAS should enable sellers to automate as many simultaneous auctions as they wish on their computer. However, only one item may be sold per auction. In particular, there are three main requirements:

- Firstly, sellers should be able auction their items using different methods and types. The auction methods being FPSB, Vickrey, English, Dutch and the auction types being Common Value and Private Value (see Section 1.2). In addition, sellers should be able to determine when their auctions start, use rational selling strategies to sell their items and determine the increase and decrease in asking price for English and Dutch auctions. (Where selling strategies are rules that determine the starting prices and reserve prices for items). Moreover, sellers should be able to test and choose their selling strategies using an auction simulator;

- Secondly, sellers should be able to advertise their location (address) in the auction system;

- Thirdly, sellers should allow other users (bidders) to check what items they plan to auction and provide bidders with an estimated sale price[2] before they register to bid in the auction. They should be able to monitor all bidders whether they are checking auctions or bidding in them.

---

2 In this auction system, sellers may decide to set reserve prices that are higher or lower than their estimated sale price, see Section 6.2.1.

Automating Bidding

The auction system should enable bidders to find sellers' auctions, check which items are up for auction and bid for their chosen item using rational bidding strategies (see 4th bullet point). However, since seller auctions are held on computers that are distributed over the Internet, bidders must first be able to find these auctions, check items for sale and finally decide whether to bid in the auctions or not. In particular, there are four main requirements:

- Firstly, bidders should be able to specify what item they would like to buy, under which auction method and at which auction sites. Moreover, bidders should be able to check: auction sites for items to be auctioned, their estimated prices, when the auctions start and what auction methods are going to be used;

- Secondly, bidders should be able to explore other auctions within the auction system that may be unknown to it. Initially, bidders may only know a few sellers (auction sites) in the auction system. However, if bidders co-operate they can share their knowledge of where sellers are located within the auction system, i.e. their addresses. Bidders should be able to co-operate in two ways. Firstly, every auction site they check, they must disclose their own auction site address (stored in their schedules) to the auction sites. Secondly, when a bidder requires the address of a new auction site, it can ask within the auction system for a new auction site address. This mechanism helps bidders advertise their own auction sites and find new auction sites. This is especially useful when the number of users acting as sellers in the auction system is large;

- Thirdly, bidders should be able to decide which seller auction (at an auction site) to register and bid in. The bidder should have control over when to stop looking for auction sites in three ways. Firstly, bidders should be able to specify when to stop looking for new auction sites. Secondly, they should be able to specify when to stop immediately at a good auction, i.e. auctions where the item up for auction has a low estimated price advertised by the seller. Thirdly, they should be able to specify that they are only interested in auctions that start before a certain date and time. After bidders have stopped searching and checking for auctions, they should make a decision whether to register with an auction. If an auction is auctioning the right item using the right method at the right time then it should choose the auction that advertised the lowest estimated price. If there are no suitable seller auctions, bidders should give up. Once bidders have decided to join an auction by registering, they cannot change their minds and must bid in that auction. However, they are free to search and check for another auction after the current one has finished. The bidders that lose the auction

should be able to bid in other auctions. Whilst the bidder that has won should stop searching for auctions and be able to assess its performance;

- Fourthly, bidders should be able to bid in the auctions using bidding strategies. Similarly, to sellers, bidders should be able to test and choose their bidding strategies. Where bidding strategies are rules that determine their maximum bids and scale-downs for items (see Section 1.2 for definitions). Therefore, the auction system should enable bidders to simulate auctions and allow them to evolve and use the best bidding strategies in real auctions held within the auction system

The last two sections described what users acting as sellers and/or bidders require from an auction system. The next section describes how these requirements are implemented using various methods and technologies. In particular, Seller agents are used as basis for automating selling within auctions and Bidder agents are used as a basis for automating the searching for and bidding in these automated auctions.

5.1.2 Design Components

The IAS enables users to automate auctions by creating Seller and Bidder agents. Seller agents are stationary and do not move. Users create and run them locally on their computer. It is the Seller agent running that automates the selling part of the auction process. Conversely, Bidder agents are mobile. They can move from one computer to the next using the Internet. Users create and run Bidder agents on their computers. However, as Bidder agents move from computer to computer, they run locally. It is the Bidder agent running that automates the searching for and bidding in remote auctions. Figure 5.1 shows the three main aspects of the IAS.

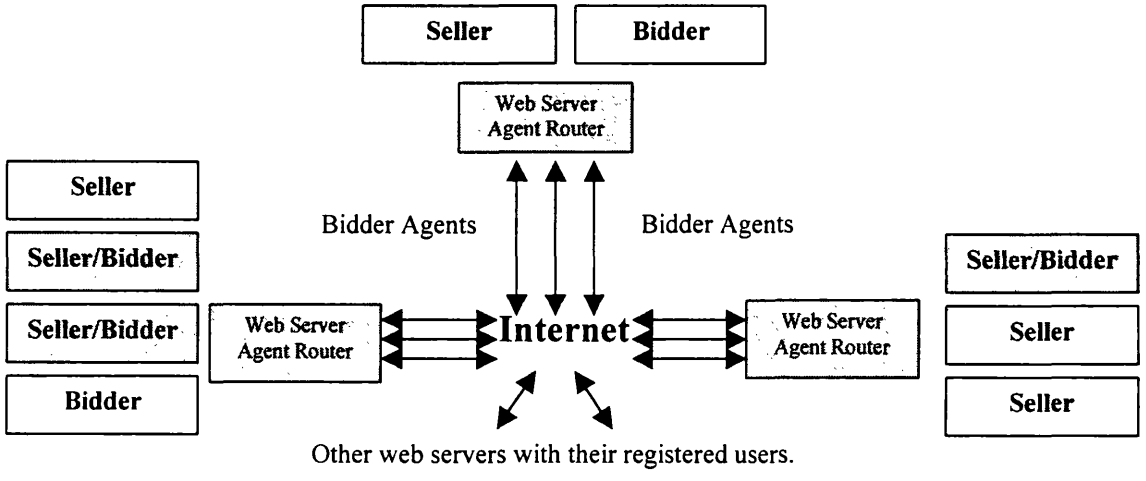


Figure 5.1 The IAS consisting of user auction sites acting as bidders and/or sellers

Firstly, each box labelled *Seller and/or Bidder* represents the software required by users to join the IAS from an auction site and create agents. Some users may just want to sell items and others may just want to buy. However, any user can do both simultaneously (they may even send Bidder agents to their own auctions). In theory, the number of users (auction sites) in the IAS and the number of Seller and Bidders agents they can create are unlimited. Secondly, distributed users are connected to each other via the Internet and web servers. The Bidder agents use the Internet to move from one computer to the next (arrows show Bidder agent flow). These connections are transient and can close whenever users decide to leave the IAS. Thirdly, users are registered with one web server that routes all Bidder agents to their auction sites. Users being grouped near their web server represent this.

However, the IAS is specified by breaking it down into separate domains according to some project design methods described in [SHA92], [SOM96]. The implementation domain is defined as all the software and hardware components that are required by the IAS but are independent of it. The Application Domain is the set of software components that are used to build the IAS in conjunction with the Implementation Domain.

The Implementation Domain

To meet the requirements laid out in the last section, the IAS uses a domain of network protocols and languages. The network protocols are the Transmission Control Protocol/Internet Protocol (TCP/IP), Hypertext Transfer Protocol (HTTP) and sockets [SPA96]. The languages used are Java [GOS95] and Perl [WAL92]. The Internet, web servers, and sockets are used to connect one user's computer to another, as described in Section 3.1.2. This enables Bidder agents to move from one user to the next. Figure 5.2 shows a Bidder agent moving from one user auction site to another (via the destination's web server that routes the Bidder agent) on a remote computer.

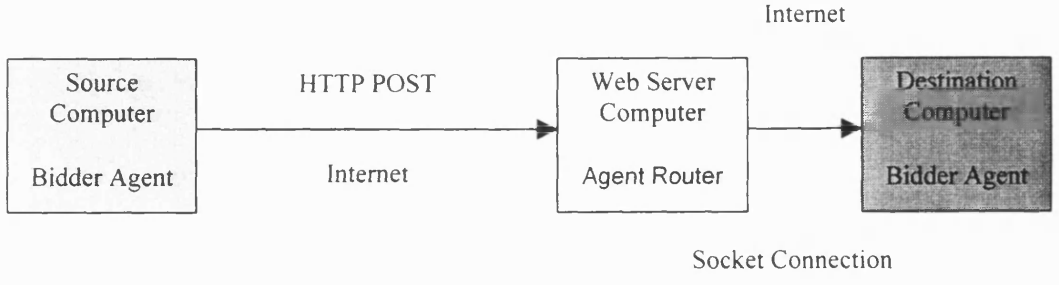


Figure 5.2 Routing agents

Sockets could be used exclusively to connect one computer to form a client/server system. However, using additional web servers had three advantages. Firstly, they already form a global network of servers that the IAS would benefit from utilising. Secondly, having a second server

enables users to be mobile. Thirdly, the security features of HTTP could be used. Since HTTP is a stateless protocol (once a request is made and carried out, the connection closes), conversations between the source computer and the destination's web server would be difficult. However, since Bidder agents move to their destinations and run locally, there is little need for inter-agent conversation for agents on different computers. They only communicate when they are on the same computer.

The implementation languages are Java and Perl. Java is used as the main development language because it is platform independent and has various features that make distributed multithreaded applications easier to implement. Perl is used to implement the Common Gateway Interface (CGI) program because, primarily, this program is used to manipulate string data and Perl is particularly good at this.

The Application Domain

Every user that wants to join the IAS and create Seller and Bidder agents to automate auctions requires application software[3]. This software needs to be installed on every user's computer and run every time they want to join the IAS. All users run their application software to form the distributed IAS. Figure 5.3 shows the components that make up a user's application software.

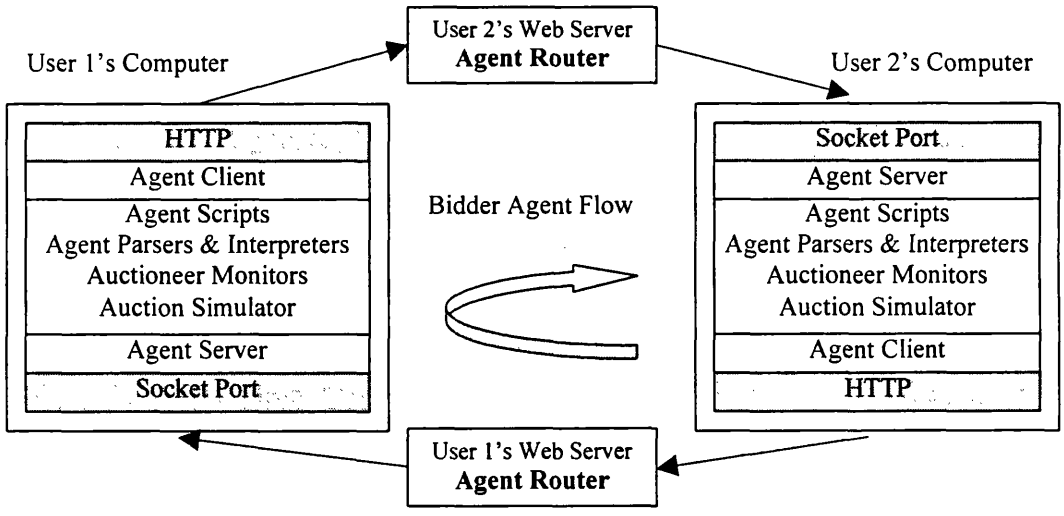


Figure 5.3 Application software components

3 The application software consists of approximately eight thousand lines of Java code. Therefore, for most components, only partial source code listings are given in the appendices.

Most components run on the user's computer whilst the agent router component is run on the user's web server. Since the agent router is used to route Bidder agents to any number of separate users, it would usually reside on a separate dedicated computer running the web server. Most components are object based except the agent scripts; i.e. the components consist of one or more objects. However, the application software is explained in terms of the following components.

Agent Client Managers

The agent client manager enables users to log into and out of the IAS, to create, monitor, and utilise their agents. When users run their application software, a browser-like GUI appears on their screen, see figure 5.4.

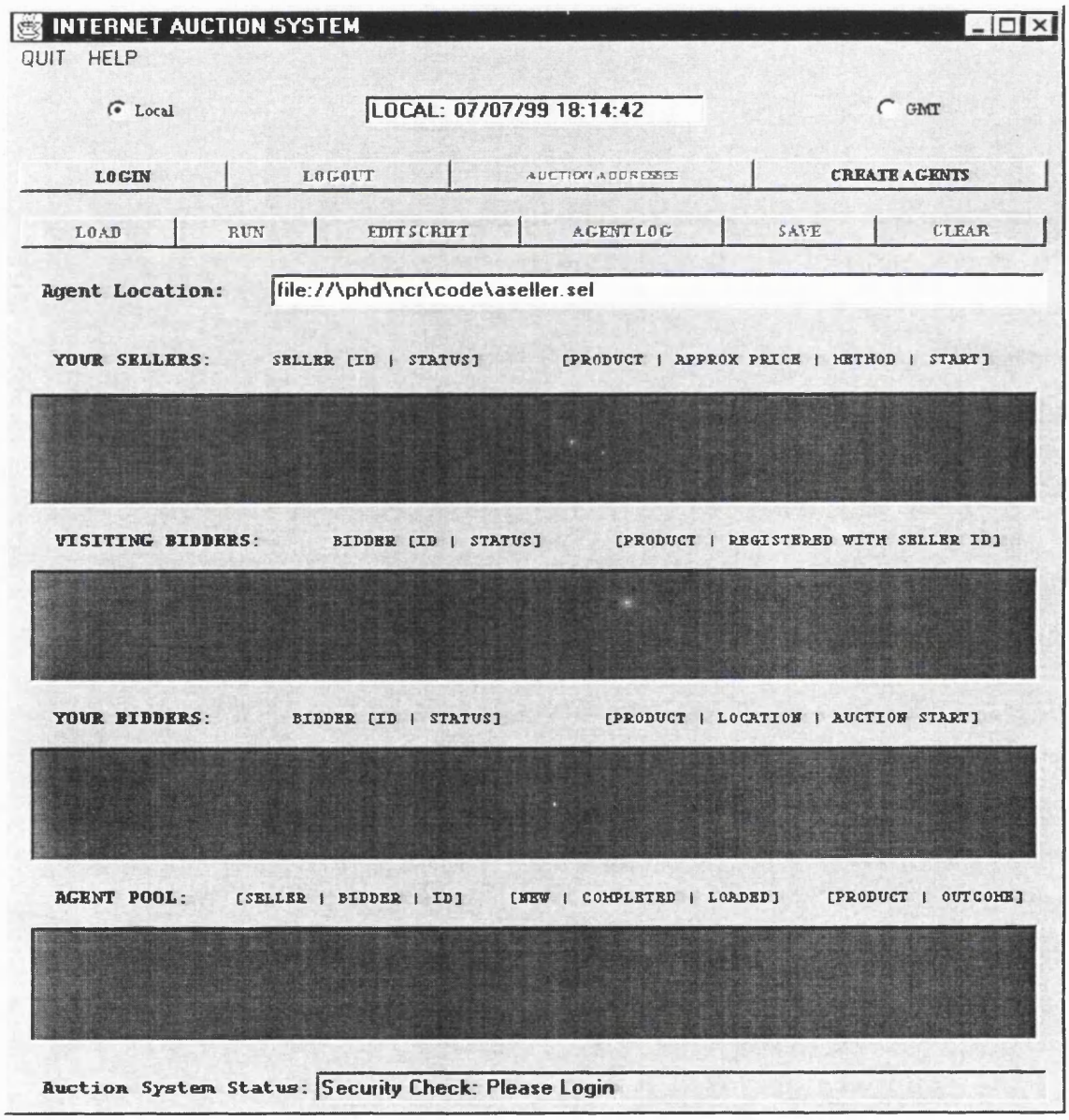


Figure 5.4. The Internet Auction System GUI

Before a user can create agents, they have to log into the IAS system. The user presses the 'login' button and another GUI appears. This GUI has the following eight fields and two options:

- Username and password;
- User Computer's IP or Domain Name Server (DNS) address and agent server's local port;
- Web server's IP or DNS address and its agent router relative path;
- Proxy server's IP address and its HTTP Telnet port;
- Filter and auto-load agents options.

Users must supply their usernames and passwords every time they log into the IAS. However, a configuration file is used to automatically fill in the other fields. The username does not have to be unique but username *and* address must be. For example, a username and address in the IAS might be *kyzyl@this.co.uk* and another *kyzyl@that.co.uk*. The agent server port will be explained in the section, Agent Server. Unlike the user's address, the web server address is fixed. Since users may log into the IAS from any computer, their computer addresses may change. In order for users to create Bidder agents that find these mobile users, they have to send them to their destination's web server. The web server redirects them to the user's current location. So all users associated with a particular web server have different names but the same address. For example, assume users *A*, *B*, and *C* share the same web server with address *routeabc.co.uk*. If users wants their Bidder agents to visit *A*, *B* or *C* they use the addresses *A@routeabc.co.uk*, *B@routeanc.co.uk*, or *C@routeanc.co.uk*. The agent router that redirects Bidder agents is located in a special directory on the web server. This can be set using the agent router's relative path field. However, this field is set to */cgi-shl/agent_router.pl* as a default. The last two fields are used if users are connected to the Internet via a Proxy server. In this case, users within the firewall may form an IAS but users outside cannot join their IAS unless the Proxy server is configured to accept agents. The 'agent filtering' option enables users to filter agents, i.e. a Bidder agent that attempts to move to a user's computer requires the user's permission. Finally, the 'auto-load agents' option enables users to automatically load agents when they log in.

Once a user has entered and confirmed this information, it is passed to their agent router for security clearance. The agent router checks the username and password. If the user is cleared, the agent router updates the user's present address so that it can route subsequent Bidder agents. The user may now use the agent manager to create and monitor their agents. However, users may only log in once per session. In addition, users may log out at any time, though, they should not log out when agents are running. When they log out their agent router is informed. In turn, the agent router informs any Bidder agents attempting to visit that the user is no longer available, see agent router source code in Appendix D.

Creating Agents

Users may create as many Bidder and Seller agents as they wish. They do this in two stages. In the first stage, users evolve their agent's bidding or selling strategies in an auction simulator (see sections 5.2, 6.2.1, and 7.2.1). Once users have evolved bidding or selling strategies, they can be embedded into their agents' scripts. In the second stage, users configure their agents ready for real auctions held within the IAS using GUI(s) (details in next two chapters). Their agents' scripts are automatically generated and are ready to run.

Monitoring and Utilising Agents

If a user runs a Seller agent, it automates the selling part of the auction process on the user's computer. Similarly, if a user runs a Bidder agent, it automates the buying process by searching for remote auctions and bidding in them. During these processes, users need to see what their agents are doing. The agent manager enables users to monitor their agents in three ways. Firstly, their agent managers have four GUI windows. The first window shows what their Seller agents are doing. The second window shows what any visiting Bidder agents are doing. The third window shows what their remote Bidder agents are doing and the fourth window shows a list of their completed or non-running agents see figure 5.4. Secondly, a user may view various log files. Each local and remote agent writes to a log file that is stored on a user's local directory. They may view any one of their agent's log files at any time. Thirdly, when an agent has completed running, a user may view its script. This contains various details on what happened to the agent during its lifetime.

Besides creating, running, and monitoring agents, users may save, load, edit, and parse them. Once an agent has been created, a user can save it to a local directory. Conversely, they may load an agent script from a local directory or even download an agent script from a remote web server. Once an agent script has been newly created, loaded or is currently loaded but non-running, users may edit and parse it. This is useful if they want to reuse the agent. Details on agent utilities can be found in Section 3.2.2.

Agent Servers

Every user that is logged into the IAS has an agent server that continually listens on a dedicated port for incoming Bidder agents that have been routed by the agent router. If users are logging in from the same computer, they must have their agent servers listening on different ports. Visiting Bidder agents could either be from other users looking at their auction site for auctions or one of

the user's own returning Bidder agents. If they are from another user, the Bidder agent normally asks the agent server for permission to visit. If the agent filter option were not activated then the agent server would allow the Bidder agent to move to the user's computer and run locally. If the agent filter option was activated then the user is warned that a Bidder agent wants to visit. The user must then respond, otherwise, the Bidder agent is refused permission to visit. See Appendix E for the agent server's source code.

Agent Scripts

Though the next two chapters describe Seller and Bidder agents in detail, they are introduced over the next few sections. Agents are ASCII text-based scripts. The Seller agent has its various features marked up using a script language derived from SGML, called Seller Agent Mark-up Language (SAML). In particular, the following five components of a Seller agent are marked up using tags:

- Identification, e.g. creator's name, address and time it was created;
- Auction Assumptions, e.g. item for sale, auction method to use, when auction should start;
- Fuzzy Set Partitioning;
- Genotype Encoding Selling Strategies;
- Memory and States.

Similarly, the Bidder agent has its various features marked up using a script language derived from SGML, called Bidder Agent Mark-up Language (BAML). However, Bidder agents have additional components marked up and they possess bidding strategies. Bidder agents require a list of auction sites to visit so that they can search for auctions within the IAS. Therefore, Bidder agents also require:

- Schedule or list of auction sites (user addresses);
- Genotype Encoding Bidding Strategies.

As mentioned earlier, users do not have to write any agent scripts because they are generated automatically. However, if they do, each user may use agent parsers to check their agent scripts.

Agent Parsers

When users modify their agent scripts in an editor, they need to parse them using agent parsers. These check the scripts lexically, syntactically and semantically. In other words, they check the

scripts have the correct tags, that they are in the correct order and the data marked up by the tags is of the correct type. However, even if the script languages have been defined and the corresponding parsers have been written for them, there is still no guarantee that the scripts will run properly. Therefore, it is better that the agent system automatically generates the scripts before they are run using an agent interpreter, see Appendix F for agent parsers' (tag checkers) source code.

Agent Interpreters

Agents automate auctions by their scripts being interpreted. Every time a user creates an agent and runs it or a Bidder agent visits from another user, a Bidder agent interpreter object is created, see Appendix G for agent interpreters' source code. This object inherits all the necessary functionality from the Java thread class so that the object can be run as a thread. The agent script is then passed to the (agent interpreter) object's constructor as a text string and executed as a separate thread [FLA96]. Therefore, each agent runs independently as a thread on a user's computer. Agents are interpreted so that they behave as finite automata. Each agent has one 'start' state, a few intermediate states, and one 'terminating' state. As agents run and interact with their environment, they change state and perform actions. The state transition diagrams for both Bidder and Seller agents can be found in the Sections 6.3 and 7.3.

However, generally the Seller agent passes through the following states: STARTING, PREPARING, AUCTIONING, and COMPLETING. During the STARTING state, a Seller agent waits until its auction is due to start. As the Seller agent waits, it accepts Bidder agents who visit and want to register to join the auction. During the PREPARING state, the Seller agent sets its starting and reserve prices (using evolved strategies) and starts the auction. During the AUCTIONING state, the Seller agent sets its asking bids and accepts the bids from the Bidder agents. Lastly, during the COMPLETING state, the Seller agent checks to see if there is a sale, and if there is, which Bidder agent won the auction? The Seller agent then stops running as a thread.

The Bidder agent is more complex and goes through the following states: STARTING, MOVING, OBSERVING, DECIDING, REGISTERING, BIDDING, COMPLETING, HOME and COOPERATING. In brief, the Bidder agent starts by visiting users (auction sites) listed in its schedule that may be holding auctions; i.e. they may have created Seller agents. On visiting an auction site, it must disclose its own list of auction site addresses. In addition, it stores information (by modifying its own script) on what it finds there. If it completes its schedule, it can ask its current auction site location for a new auction site address. After a certain time, the

Bidder agent stops searching and makes a decision whether to attend an auction or not. If it does, it must register with the Seller agent and wait until the auction starts. Otherwise, the Bidder agent returns home. During the BIDDING state, the Bidder agent bids and competes with the other registered Bidder agents. If the Bidder agent wins the auction, it returns home. Otherwise, it can decide to register at another auction. Eventually, the Bidder agent returns home after either winning one auction or losing all the auctions it attends.

Auctioneer Monitor

Although Seller agents automate auctions and Bidder agents bid in them, they do not actually communicate with each other directly. Instead, all communication between Bidder agents and a Seller agent is mediated by an auctioneer monitor.

Applications like Internet auctions require agents to run concurrently. Therefore, on every computer within the IAS, all agents run concurrently as separate threads. However, this creates a multithreaded environment that needs to be controlled [RUS92]. Auctioneer monitors are used to control agent threads as follows. When a Seller agent is first run, it creates an auctioneer monitor object. Every Seller agent has one and only one auctioneer monitor associated with it. However, since a user may run several Seller agents simultaneously, there may be many auctioneer monitor objects in use. The auctioneer monitor has four main uses. Firstly, it enables a Seller agent to advertise its prospective auction, i.e. item for sale and its estimated sale price. Secondly, when Bidder agents visit an auction site, they check each Seller agent's auctioneer monitor for various details, e.g. item to be auctioned, estimated price, auction's start time etc. Thirdly, if a Bidder agent decides to participate in a particular Seller agent's auction, it registers with the Seller agent's auctioneer monitor. Fourthly, it mediates all communications between the Seller agent and all the Bidder agents during the actual auction process for each of the four auction methods see Appendix C for auctioneer monitor's source code.

Auctioneer monitors are used to automate auctions for four main reasons. Firstly, since Bidder agents are mobile, they can move to the auction site and communicate with the Seller agents locally; i.e. no fixed connection is needed. This reduces inter-agent network traffic and the complexity associated with distributed agent communications. Secondly, they make temporal counter-speculation (or arbitrage in which one agent receives information before another) impossible because all agents are strictly controlled and synchronised. Agents that receive information before others could have an unfair advantage especially in a Dutch auction [BIN96]. Thirdly, they can prevent problems associated with concurrent applications like busy waiting and deadlock by implementing solutions like semaphores and 'barriers' [COH96]. Fourthly, they

enable auctions to be automated faster since all the processing takes place locally. However, agent monitors have drawbacks. In particular, programming monitors that control multithreaded applications is difficult. Moreover, it is difficult to verify and guarantee that they work properly.

Initialisation

Seller agents create auctioneer monitor objects and supply the following essential details to its constructor. These are then allocated to the auctioneer monitor's shared-variables:

- Seller Agent's ID;
- Auction Method;
- Auction Start Date and Time;
- Item up for Auction;
- Seller's Estimated Sale Price or Valuation for the Item.

During a Bidder agent's visit to a user's auction site, it checks a Seller agent's auctioneer monitor by reading its shared-variables. It may do this at many auction sites. Eventually it may decide to join and register at a Seller agent auction. The Bidder agent registers by writing its identification details to one of the Seller agent's auctioneer monitor shared-variables. The Seller agent can then check its auctioneer monitor to see which Bidder agents have registered. However, since all the auctioneer monitors are accessible by all other agents, the shared-variables are synchronised. This prevents one agent reading a variable whilst another is writing. This is critical for the multithreaded auctions because one Seller agent and any number of Bidder agents automate the auctions by reading from and writing to shared-variables. The multithreaded auctions are now described for the four auction methods.

The Multithreaded Auctions

Auctioneers must be fair if bidders are going to trust the auction process [AGO96]. This is particularly true of sealed-bid auctions. It is assumed that the auctioneer monitor is fair and impartial to the registered agents. The four auction methods are now described in terms of how the auctioneer monitor controls the Seller agent and Bidder agents threads, see Appendix C.

Sealed-Bid Auctions

In a sealed-bid auction, the Seller agent notifies all registered Bidder agents that the auction has begun by setting a flag (synchronised shared-variable) within its auctioneer monitor. It then

waits. However, Bidder agents may attempt to read this flag before it has been set by the Seller agent. If a Bidder agent does this, its thread is sent into a waiting state. Since only one Bidder agent may check the flag at any one time, any number of Bidder agent threads could be waiting. When the Seller agent sets the flag, all the waiting Bidder agent threads are woken up (using a Java thread notify method call [COH96]). They now, one at a time, read the flag and prepare for the auction. After all the Bidder agents have read the flag, they submit their bids. The last Bidder agent to submit its bid causes a notification to be sent to the Seller agent to wake it up. The Bidder agents now wait for the outcome of the auction. The Seller agent analyses the submitted bids. If the highest bid is greater than its reserve price, that bid wins. The Seller agent sets another shared-variable with details of the auction's outcome ready for the Bidder agents to read. This again notifies all the Bidder agents that the Seller agent has completed the auction. The Bidder agents can then read the auction outcome. This consists of whether there was a sale or no sale. If there was a sale, the shared-variable also stores who won and the winning bid price.

English Auction

The English auction multithreading process starts the same way as the sealed-bid auction process. The Seller agent must confirm that the auction has started. It then calculates its starting and reserve prices (may be different to the item's estimated sale price). All the Bidder agents check the flag to find out whether the auction has started. However, once the auction has started, the auction process is different to sealed-bid auctions. The Seller agent waits for all Bidder agents to check that the auction has started. It then sets the asking bid to its starting price by setting the asking bid shared-variable in the auctioneer monitor. In doing this, all the waiting Bidder agents are woken up (using a Java thread notify method call). The Bidder agents now attempt to read the asking bid variable one at a time. After all the Bidder agents have read the asking bid, they reply with either 'Y' (accept bid) or 'N' (do not accept bid) by setting the bid answer(s) shared-variables, again one at a time. After the Bidder agents have done this, they wait. In doing this, the waiting Seller agent is woken up. The Seller agent then reads the bid answers. If there are two or more 'Y' bid answers, the process happens again in four distinct cycles, as follows:

Cycle 1 - Seller agent sets new higher asking bid, waits and wakes all the Bidder agents up;

Cycle 2 - Each Bidder agent attempts to read the new asking bid. Only one Bidder agent at a time can read the asking bid. All others must wait. The Bidder agent that has just read the asking bid wakes up another Bidder agent that is waiting to read it;

Cycle 3 - The last Bidder agent to read the asking bid, sets its bid answer. Then it waits and wakes up another Bidder agent so that it can set its bid answer;

Cycle 4 - The last Bidder agent to set its bid answer wakes the Seller agent up. The Seller agent then reads each Bidder agent's bid answer.

Eventually, only one bid answer given by the Bidder agents will be 'Y'. The Seller agent then checks to see if the bid is greater than its reserve price. If it is not, the Seller agent increases the asking bid and resumes the auction process. Eventually, when the asking bid is greater than the reserve price, the Seller agent stops and assesses whether any Bidder agent accepts. If a Bidder agent does, it sets the outcome variable to 'SALE' and wakes up the waiting Bidder agents. Otherwise, it sets the outcome variable to 'NOSALE'. Finally, the Bidder agents can read the auction outcome. This consists of whether there was a sale or no sale. If there was a sale, the shared-variable also stores who won and the winning bid price.

Dutch Auction

The Dutch auction process is very similar to the English auction process. The same cycles of agents reading and writing shared-variables occurs. However, there are two differences. Firstly, asking bids decrease. Secondly, the auction process stops if either the asking bid falls below the reserve price, in which case there is no sale, or a Bidder agent answers 'Y' to an asking bid. The Bidder agent to submit the first 'Y' bid answer wins the auction.

Tied Bids

In the above auction processes, it is assumed there is only one Bidder agent with the highest bid. However, it can happen that two or more Bidder agents submit equal bids in a sealed-bid auction or two or more Bidder agents supply 'Y' bid answers in a Dutch auction. Moreover, two or more Bidder agents can be tied in English auctions by accepting an asking bid and rejecting the next increased asking bid. Tied bidding is settled by awarding the sale to the tied Bidder agent that registered first.

Agent Router

The agent router and its functionality were introduced earlier in this chapter and in detail in Chapter 3. Therefore, the message protocols between the clients, servers and agent router will not be repeated here. However, the two main reasons for using the components in the IAS are reviewed. Firstly, it enables mobile users to log in from any computer for verification. Users that have been verified may join the IAS and the agent router updates a file (held on the web server)

with the user's current address. This file (called Username.txt) stores the following information for each user registered for that particular web server:

- Username and Password;
- Current Address, i.e. computer's IP or DNS address and agent servers' port number;
- Whether they are currently logged into the IAS;
- Whether they are filtering Bidder agents.

Therefore, every web server in the IAS must have a 'Username.txt' file that include all the users that would like Bidder agents redirected to them via that web server. New users are required to add their unique names and agent server port numbers to the web server's Username.txt file. Secondly, it redirects visiting Bidder agents. When a Bidder agent attempts to visit a user, it must go through the agent router. The agent router will then form a socket connection with the correct user's agent server. Once this connection is formed, the Bidder agent may pass from its source computer to its destination computer. Once the agent script is located on its destination's computer, a Bidder agent interpreter object is created. The agent script is passed to the interpreter's constructor as a string and run locally. See Appendix D for agent router's source code.

Further work

Some features not implemented in the IAS (and recommended for further work) are transaction integrity, agent certification, encrypting agent strategies, and HTTP security features. In addition, agents are not persistent and could be lost. The IAS has no way of recovering from computer crashes or network failures. For more details see Section 9.3.

5.2 The Auction Simulator

The auction simulator developed in this thesis is used by users to evolve good bidding and selling strategies for their Bidder and Seller agents. These can then be embedded into their agent's scripts and used in real IAS auctions. The simulator's requirements, assumptions, and configurable parameters are discussed. However, for details on how auctions are modelled as games, how agents are modelled as players, and how Bidder and Seller strategies are encoded using Fuzzy-Genetic Algorithms (FGA) see Sections 6.2.1 and 7.2.1. In addition, for an evaluation of the simulator and the bidding and selling strategies evolved using it, see Chapter 8.

5.2.1 Requirements

The simulator should enable users to evolve bidding and selling strategies for the four auction methods and two auction types (Private value and Common value). In this thesis, an auction type can be considered an assumption since real auctions rarely consist of bidders who all know their private-values precisely or who all value an item identically, see Section 1.2. However, as mentioned in the introduction they are simpler to model and there are some theoretical results available to check results obtained by using the simulator.

Simulated Seller Agents

The simulator uses simulated Seller agents (simulated agent) to model the selling aspects of an auction. These simulated agents only possess selling strategies but otherwise, they are the same as Seller agents used in real auctions. In any case, simulated and real Seller agents have their decision making limited in their auctions. Seller agents only need make at most two decisions in simulated or real auctions. In sealed-bid auctions, the only decision Seller agents make is what to set as the reserve price? Whilst in English and Dutch auctions, Seller agents make two decisions: what to set for the starting and reserve prices? In order for agents to make a decision, they require information. This could come from three sources. Firstly, from beliefs about the auctioned item's value[4]. Secondly, from bidding in an auction. Thirdly, from previous auctions. However, in these simulated auctions the third information source is ignored. Moreover, the Seller agent must decide what the starting price and reserve prices are before the bidding starts. Therefore, the only information used by the Seller agent to determine starting and reserve prices is from prior beliefs about the value of the auctioned item given to it by its user. The Seller agent's beliefs are:

- Expected Bidder Valuation;
- Bidder Valuation Range;
- Expected Bidder Scale-down (for CV auctions only).

All other information is ignored; i.e. the Seller agents are risk neutral and only use these three pieces of information to make their decisions.

Simulated Bidder Agents

The simulator uses simulated Bidder agents to model the bidding process within auctions. These simulated Bidder agents only possess bidding strategies but otherwise they are the same as

[4] If the Seller agent assumes a PV type auction then it knows the value precisely.

Bidder agents used in real auctions. However, the auction type affects the decisions they have to make. In PV auctions, they know their own private values. Therefore, to prevent making a loss, they should not bid above their own private value. However, in CV auctions, Bidder agents do not know the precise value of the auctioned item. They may over bid and induce a loss. Since all the Bidder agents are estimating the value of the item, the winner is the one who over values the item the most and bids the highest. This should worry the winning Bidder agent because it is likely that it has over valued the item, over bid, and made a loss. This is called the 'winner's curse'. Bidder agents can avoid this if they scale down their bids to avoid over bidding. Therefore, in CV auctions, Bidder agents should make a second decision on how much to scale down their bid. This is called a Bidder's scale-down, see Section 1.2 for further details [RAS96], [VIC61]. Similarly, a Bidder agent could use three sources of information to make bidding and scale-down decisions. It could use information supplied by its user. It could use information gained from other Bidder agents (or bidders) during the auction or it could use information from previous auctions. In these simulated auctions, Bidder agents use prior information supplied to them by their users and posterior information gained from bidding in the auctions (only applicable in real English IAS auctions). Therefore, the only information used by the Bidder agent to determine it bid and scale-down are:

- Expected Bidder Valuation;
- Bidder Valuation Range (maximum valuation – minimum valuation);
- Expected Bidder Scale-down (for CV auctions only);
- Current Asking Bid (for English auctions only).

During the English auction, Bidder agents crudely re-estimate 'expected bidder valuation' and 'bidder valuation range'. All other information is ignored. The Bidder agent is risk neutral and only uses these four pieces of information to make their decisions at any point during the auction.

5.2.2 Design

The simulator's design is based on ideas described in Section 4.2. This explained that if an agent environment could be modelled as a game in which the agents are the players then this would make the modelling simpler. The simulator treats the various auction methods and types as games with implicit assumptions. In particular, the simulator's design consists of simulated agents and a simulated auction environment. Moreover, both Seller agents' and Bidder agents' sources of information are partitioned using fuzzy sets and FGA(s) are used to encode and evolve their strategies during the simulated auctions. The simulator has a GUI interface that enables

users to configure auctions so that they can evolve bidding and/or selling strategies for their chosen auction environments (see table 5.1).

Auction Environment Parameters
▪ Auction scenario ▪ Auction method ▪ Auction type ▪ Agents' prior beliefs or information parameters Minimum/maximum expected bidder valuations Minimum/maximum bidder valuation ranges Minimum/maximum expected bidder scale-downs (CV auctions only) Private value (PV auctions only)
Fuzzy Partition Parameters
▪ Number of fuzzy sets used to partition 'expected bidder valuation' ▪ Number of fuzzy sets used to partition 'bidder valuation range' ▪ Number of fuzzy sets used to partition 'expected bidder scale-down'
Genetic Algorithm's Parameters
▪ Number of generations ▪ Number of Sellers agents in an evolving population ▪ Number of Bidder agents in an evolving population ▪ Number of fixed strategy bidder opponents ▪ Auctions to play per round ▪ Crossover probability ▪ Mutation probability ▪ Agent's credit ▪ Phenotype Gene alleles for Seller agent's Starting Price decision chromosome Gene alleles for Seller agent's Reserve Price decision chromosome Gene alleles for Bidder agent's Bidding decision chromosome Gene alleles for Seller agent's Scale-down decision chromosome

Table 5.1 Auction simulator and Fuzzy-Genetic Algorithm parameters

Users may configure the simulator to simulate various auctions using the parameters. The first group enables users to determine the auction environment. These are described in five issues. The first issue concerns which auction methods and types can be simulated. The second issue concerns the fairness of auctions. The third issue concerns agents having fixed or evolving strategies. The fourth issue concerns an agent's prior beliefs. The fifth issue concerns the quantity of items for sale per auction.

1. The simulator simulates eight auction method/type combinations, i.e. FPSB, Vickrey, English, and Dutch methods that are either Private Value or Common Value.

2. Auctions are fair. All Bidder agents bid independently. There is no Bidder agent collusion or syndication. Seller agents are fair as well; i.e. there are no auctioneer tricks or dishonesty. This is particularly important in the sealed-bid auction in which only the auctioneer would know which bidder has really won [AGO96].

3. This simulator can model two auction scenarios. Scenario 1: in every auction *one* Bidder agent has evolving strategies, all other competing bidders have fixed strategies and the seller has fixed strategies. For each auction, all its bidder opponents have fixed values for their bids and scale-downs randomly selected from uniform distributions. In addition, the seller has fixed values for its starting and reserve prices such that the item is always sold, i.e. reserve price is set to zero, and starting prices set low or high enough in English and Dutch auctions. Scenario 2: in every auction, the Seller agent has evolving strategies and *all* bidders have fixed strategies. For each auction, all bidders have fixed values for their bids and scale-downs randomly selected from uniform distributions. Auction scenarios are discussed in the next two chapters.

4. The purpose of using a FGA is to evolve good bidding and selling strategies from random ones. In other words, the agents learn or adapt to do well in their auction environment. However, agents make decisions primarily on their beliefs about other bidders' valuations. This can lead to the following problems. An evolving agent could have incorrect beliefs about other bidders' valuations but good strategies, i.e. if only the agent had the correct beliefs it would do well. This could cause the agent to do poorly or well, but for the wrong reasons. In another situation, an evolving agent could have incorrect beliefs and poor strategies and still do well. Again, for the wrong reasons. Therefore, in these simulations (but not in real IAS auctions), it is assumed that evolving agents have correct prior beliefs about their fixed strategy bidder opponents' valuations. They know the 'expected bidder valuation', 'bidder valuation range' and 'expected bidder scale-down' of their bidder opponents. Therefore, if an agent does well, it is because it has good strategies.

5. In a simulated auction, only one item is auctioned and its type is unimportant.

The additional parameters enable uses to configure the FGA (partition the agents' information spaces). These are described throughout the next two chapters. However, the GUI also enables users to observe how the agents perform in the simulated auctions. Various statistics are produced that summarise the agents' performances. The user can then select the best strategies for their agents to use in real IAS auctions.

5.3 Achievements

The IAS had two main requirements. Firstly, it should enable users to automate the selling process in auctions. Secondly, it should enable users to automate the buying process in these automated electronic auctions. The buying process includes both seeking the auctions and bidding in them. The basis for automating both selling and buying was to use software agents. Adaptable Seller agents were used to automate selling in auctions held on a user's computer. Adaptable Mobile (Bidder) Agents were used to automate the searching for and bidding in remote auctions held by Seller agents.

To implement a novel IAS, a mobile agent infrastructure for the Bidder agents to move around in was designed together with agent interpreters, parsers, routers, auctioneer monitors and auction simulators. The agents were interpreted as finite automata and run as threads forming multithreaded auctions. The auctioneer monitor controlled these multithreaded auctions by prioritising and synchronising the agent threads. In addition, both Bidder and Seller agents were adaptable. Users could use their auction simulators to evolve their agents' bidding and selling strategies for all auction methods and types. Users could also configure the simulated auctions to reflect their own preferences and valuations using parameters. In particular, users could configure and use the FGA to encode and evolve agent strategies by testing them in rounds of simulated auctions. It was hoped that the FGA would evolve better strategies for each successive generation. Once the FGA had finished evolving strategies, users could select strategies for their agents to use in real IAS auctions.

Chapter 6 – The Seller Agent

The last chapter introduced the main requirements for automating the selling process in Internet auctions using Seller agents. This chapter completes the description of Seller agents in three sections. The first section describes how agent strategies are encoded using genetic structures, how fuzzy logic is used to summarise an auction's state, and how Seller agent strategies are adapted in an auction simulator. The second section describes how Seller agents are scripted using SGML. The third section describes how Seller agents are implemented as automata. This explains the state transitions that Seller agents pass through to automate the selling process. The chapter's last section describes some useful utilities that users can use to manage their agents and view their auctions.

6.1 Overview

The following diagram (figure 6.1) symbolically shows the various components that make up a Seller agent. These are explained throughout this chapter. However, they are briefly introduced in this section to give some indication of how Seller agents are implemented within the Internet Auction System (IAS).

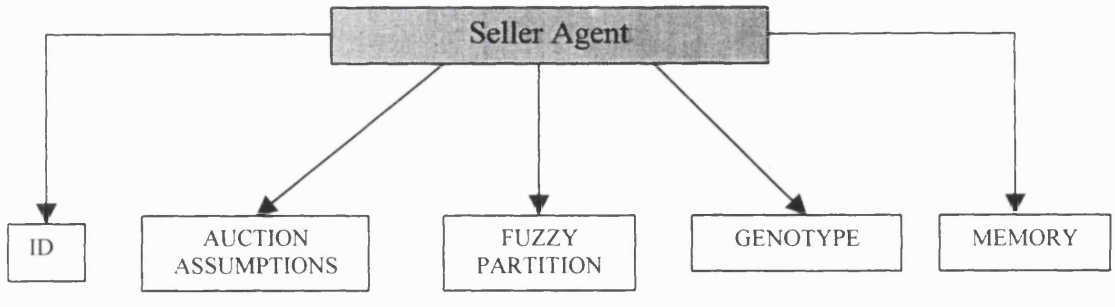


Figure 6.1 The Seller agent

A Seller agent has five structured components. Its first component is IDENTIFICATION (ID). Every Seller agent should have a unique ID that distinguishes it from all other Seller agents in the IAS. A Seller agent's ID is made up of its user's (creator's) name, creator's address and the date and time it was created. Its second component is AUCTION ASSUMPTIONS. When users create Seller agents, they supply various pieces of information, e.g. auction type, auction method, and beliefs about bidder valuations. These are used by the Seller agent to configure its auction. Its third component is FUZZY PARTITION. This stores information on how the Seller agent's information space is partitioned using fuzzy sets. Its fourth component is GENOTYPE. This

stores the Seller agent's strategies (for setting starting and reserve prices in the auctions) in genetic structures called chromosomes and genes. Its final component is MEMORY. This stores all the information that is gained by the Seller agent once it starts running. This includes Bidder agents that registered to join its auction, bidding information gained from the auction and the Seller agent's state. These components are now described in the context of how users utilise their Seller agents and how they are implemented in the IAS.

6.2 Utilisation and Implementation

Users can create their Seller agents in three ways. Firstly, users can create new agents using GUI(s); i.e. forms are used to configure the agent. Secondly, users can load agents from local file directories or from web servers. Thirdly, users can reuse an existing Seller agent that has already been used in the IAS. However, the remainder of this section assumes that users create new Seller agents.

Users create new Seller agents in three stages, see figure 6.2. In the first stage, users evolve their Seller agents' selling strategies in an auction simulator. In the second stage, users enter via GUI(s) assumptions about the auctions that they want to automate. In the third stage, users run and monitor their Seller agents.

STAGE 1	STAGE 2	STAGE 3
Adaptation	**Configuration**	**Automation**
Implementation:	Implementation:	Implementation:
• Auction Simulator • Auction as a Game • Seller Agent Player • Fuzzy Sets • Fuzzy-Genetic Algorithms (FGA)	• Seller Agent Scripts • Mark-up & Tagging • SGML DTD/Parser • SAML mark-up	• Finite Automata • Seller States • Interpretation • Multithreaded Auctions • Auctioneer Monitors

Figure 6.2 Utilising the Seller agent and its implementation

6.2.1 Adaptation

If users want to create new Seller agents (and not load or reuse existing agents) then, firstly, they must evolve their Seller agents' selling strategies. The auction simulator is used to adapt bidding and selling strategies. (Adapting selling strategies is described in this chapter and adapting bidding strategies is described in the next chapter on the Bidder agent). The auction simulator has three groups of parameters that need setting before the Seller agent's strategies can be evolved

(see table 5.1). The first group set an auction's assumptions, e.g. auction method used, an agent's prior beliefs or information about bidder valuations etc. The second group set the number of fuzzy sets used to partition an agent's information space. The third group set the standard parameters associated with Genetic Algorithms (GA), e.g. population sizes, number of generations, gene alleles. These parameters are described in terms of how they are used to simulate auctions and evolve selling strategies.

The Simulated Auction Game

Simulated auctions can be considered games with well-defined rules. The players are simulated Bidder and Seller agents. This section gives specific details of the strategic decisions made by Seller agents in the simulated auctions and the information they use to make those decisions. Next, the section describes how Seller agents' strategies are encoded using fuzzy sets and genetic structures. Finally, the section describes how FGA(s) are used to evolve these strategies. See Section 4.2 for general information on games and strategies.

Information and Strategic Decisions

A Seller agent's selling strategies depend on the auction method and auction type. The auction method determines the rules by which an item is auctioned. The four methods are FPSB, Vickrey, English, and Dutch. In the FPSB and Vickrey auctions, the Seller agent only needs to decide what reserve price to set on the auctioned item. However, in the English and Dutch auctions, the Seller agent needs to decide what reserve price and what starting price to set? See table 6.1. The auction types define how Bidder agents' (*not* Seller agent) value the auctioned item, see Section 1.2. However, the Seller agent still makes the same decisions for both Private Value (PV) and Common Value (CV) auction types.

Strategic Decisions	FPSB	Vickrey	English	Dutch
Private Value	Reserve Price	Reserve Price	Reserve Price Starting Price	Reserve Price Starting Price
Common Value	Reserve Price	Reserve Price	Reserve Price Starting Price	Reserve Price Starting Price

Table 6.1 Seller agent's strategic decisions

Given the auction method, type and number of Bidders agents in the auction (set as a GA parameter), the Seller agent must make some further assumptions about the prospective bidder valuations[1] for the auctioned item. To keep the modelling simple [VIC61], [RUS92], [FRI93], the Seller agent assumes that the only pieces of information it will use to make decisions on what values to set for the reserve price and starting prices are:

- Expected Bidder Valuation;
- Bidder Valuation Range (maximum valuation – minimum valuation);
- Expected Bidder Scale-down (for CV auctions only).

If the distribution of bidder valuations is assumed Uniform then stating the expected (average) and range is sufficient to completely specify the distribution. In the simulated auctions, the distribution of bidder valuations and scale-downs is assumed Uniform.

However, there are three related issues to evolving strategies using these three pieces of information. Firstly, since in the sealed-bid and Dutch auctions little or no information on bidder valuations is given away during the auction, this information represents users' prior beliefs. The prior beliefs that users give to their Seller agents may or may not be accurate. However, in the simulated auctions, the Seller agent knows the distribution of bidder valuations. This should enable the Seller agent to evolve good strategies given correct beliefs. If this were not the case then the Seller agent could evolve strategies using incorrect information. The evolved strategies would then be incorrectly used in the real IAS auctions. Secondly, if the Seller agent is going to be useful in real IAS auctions it must have a complete set of strategies for a variety of bidder valuation distributions. In real IAS auctions, the Seller agent's beliefs might be inaccurate. However, the Seller agent is required to decide its starting price and reserve price before bidding starts. Therefore, it does not have a chance to re-estimate what it believes bidder valuations to be. Even so, Seller agents have a set of strategies for various auctions with different bidder valuation distributions. This enables Seller agents to be reused in more than one auction of the same type and method. Therefore, it has strategies for all auctions where:

- Expected bidder valuation lies between a minimum and maximum;
- Bidder valuation range lies between a minimum and maximum;
- Expected bidder scale-down lies between a minimum and maximum (CV auctions).

1 Bidder agents' utilities for an item or any risk associated with the auction is ignored, i.e. they are risk neutral. Only monetary amounts are used in the modelling an agent's strategic decisions.

To evolve good strategies the Seller agent may need to participate in many auctions. Users specify the auctions that they want to evolve strategies for by setting the following parameters:

- Auction method;
- Auction type;
- Private value (PV auctions only);
- Minimum expected bidder valuation;
- Maximum expected bidder valuation;
- Minimum range for bidder valuations;
- Maximum range for bidder valuations;
- Minimum bidder scale-down (CV auctions only);
- Maximum bidder scale-down (CV auction only).

This enables the Seller agent to try out its strategies in many auctions where the bidder valuation distributions are different. For example, one auction may have bidder valuations represented by the Uniform distribution, *U[100, 150]* where the minimum bidder valuation is 100, the maximum is 150, the expected bidder valuation is 125, and the range is 50. Another auction may be represented by U[32,44] with a different expected value and range.

Fuzzy Partitioning Auction State Spaces

A Seller agent uses three pieces of information to decide what starting price and reserve price should be set in an auction. These three pieces of information are 'expected bidder valuation', 'bidder valuation range' and 'expected bidder scale-down'. It is assumed that the bidder valuation distributions are Uniform. An agent's information space consists of all the possible values for these three pieces of information. To reduce the information space (i.e. all bidder valuation distributions), fuzzy sets are used to partition it.

The fuzzy partition group of parameters enables a user to determine how many fuzzy sets will partition each of the three pieces of information. Reasons for partitioning using fuzzy sets were given in Section 4.2.2. However, the main reason is to make the task of evolving good strategies easier. A user may cover 'expected bidder valuation', 'bidder valuation range' and 'expected bidder scale-down' with two or more fuzzy sets. Since, the Seller makes it decisions before the auction has started, these three pieces of information can be considered to represent the auction's state. The Seller agent's auction state space can be represented as a three-dimensional space of possible values (see figure 6.3). In PV auctions, the third piece of information, 'expected bidder scale-down' can be considered zero; i.e. information space is two-dimensional.

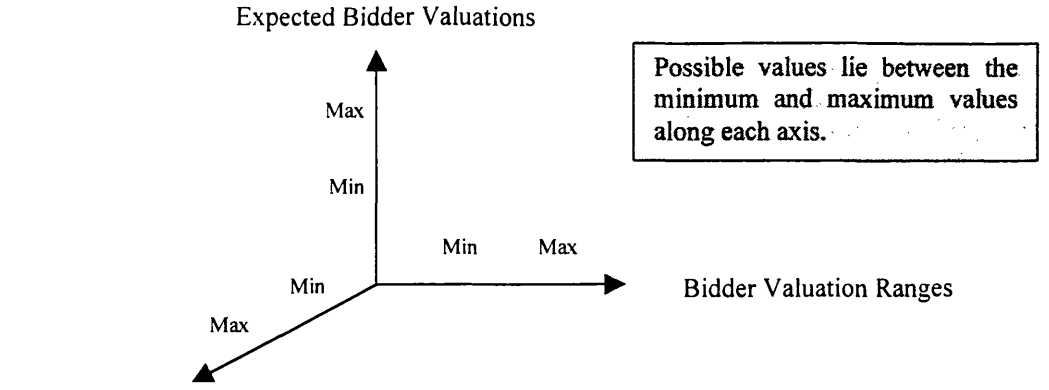


Figure 6.3 Auction state space

Therefore, an agent's information state space (or auction state space) for PV auctions can be partitioned with at least four *(2*2)* fuzzy sets because bidder scale-down is not used in PV auctions. Moreover, an agent's information state space (or auction state space) for CV auctions can be partitioned with at least eight *(2*2*2)* fuzzy sets. The finer the partition the more strategies are required. This makes the task of evolving good strategies more difficult. Allowing users to specify the partition gives them control over how long they want to spend evolving strategies and how accurate they want them to be.

Fuzzy Auction States

In Section 4.2.2, examples were given of how fuzzy sets could be used to partition state spaces. However, this section demonstrates with a specific example of how fuzzy sets are used to partition the auction state space for PV auctions. In these auctions, Seller agents only need consider two pieces of information in deciding what value to set for the reserve price. Also, it is assumed that $p1$ fuzzy sets are used to partition 'expected bidder valuation' and $p2$ fuzzy sets are used to partition 'bidder valuation range'. If the actual 'expected bidder valuation' is v and the 'bidder valuation range' is r then for each fuzzy set used in the partitioning, a corresponding membership value can be calculated. See Appendix A for the derivation of the formulae. Since only neighbouring fuzzy sets overlap, the 'expected bidder valuation' value v would only have one or two membership values > 0, see figure 6.4. Therefore, an array of $p1$ fuzzy set membership values can be used summarise the value v. In the example given in figure 6.4, $p1$=5 and the array is represented as (0,0,0.3,0.7,0). Since the fuzzy sets overlap in a particular way, this array's elements always sum to one, see Section 4.2.2. An identical procedure can be done for 'bidder valuation range' value r. However, this time it is assumed that $p2$ = 4, i.e. four fuzzy

sets partition 'bidder valuation range'. A similar array can be formed that has four elements, e.g. (0,0,0.2,0.8).

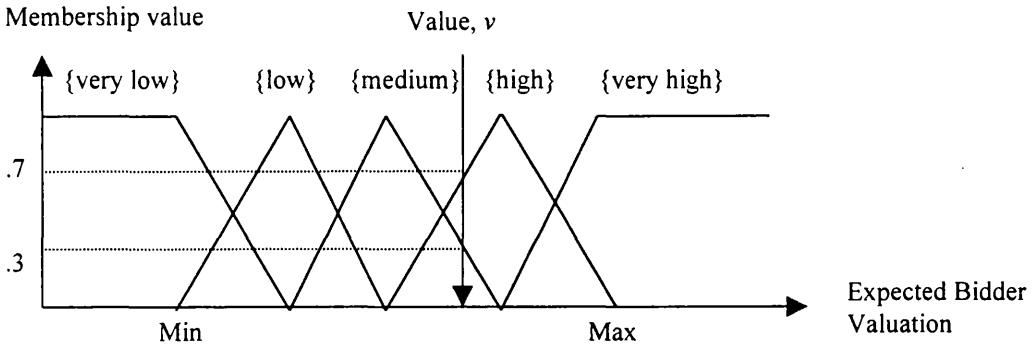


Figure 6.4 Fuzzy set partitioning of 'expected bidder valuation' using five fuzzy sets

To summarise both the 'expected bidder valuation' and the 'bidder valuation range' the two individual arrays are combined in the following way. The membership value for any combination of 'expected bidder valuation' and 'bidder valuation range' is calculated by multiplying the individual membership values. For example, the membership value for the combined fuzzy sets

{low 'expected bidder valuation' and low 'bidder valuation range'} = membership value of {low 'expected bidder valuation'} * membership value for {low 'bidder valuation range'}.

Therefore, for every combined fuzzy state a membership value can be calculated for all the fuzzy set combinations. This can be summarised in an array. If *V[i]* is the *ith* element of the first array, *R[j]* is the *jth* element of the second array then the product is defined as *F[k]:*

$$F[k=j+4*(i-1)] = V[i]*R[j] \quad \text{for } i=1,2,3,4,5 \text{ and } j=1,2,3,4 \qquad (1)$$

Therefore, if V[i] = (0,0,0.3,0.7,0) is the array for 'expected bidder valuation' and R[j] = (0,0,0.2,0.8) is the array for 'bidder valuation range' then the product F[k] = (0,0,0,0,0,0,0,0,0,0,0,0,0.06,0.14,0,0,0,0.24,0.56,0).

Most elements of *F(k)* are zero because most membership values are zero. In addition, *F(k)* is normalised in that its elements sum to one. A similar procedure can be done to summarise the Seller agent's information in a CV auction. The additional information is 'expected bidder scale-down'. Three individual arrays can be calculated and multiplied together. The resulting array would now have *p1*p2*p2* elements with each element being calculated by multiplying the individual membership values of the fuzzy sets. These arrays are used to summarise the Seller

agent's information about the state of the auction before the auction has started. The Seller agent uses these arrays to make its decisions on what values to set for the reserve and starting prices.

Decision Chromosomes

The Seller agent's genotype is made up of two chromosomes, one for each decision. The first decision is starting price (for English and Dutch auctions), the second decision is reserve price (applies to all four auction methods). Each decision chromosome consists of a number of genes. As explained in the last section an auction's state consists of two/three pieces of information, 'expected bidder valuation', 'bidders valuation range' and 'expected bidder scale-down'. Fuzzy sets are used to partition information or auction state space using a finite number of fuzzy sets. Each gene maps to one particular combination of fuzzy sets or fuzzy state. Therefore, if five fuzzy sets were used to partition 'expected bidder valuation' and four fuzzy sets were used to partition the other two pieces of information then there would be (5*4*4) eighty fuzzy states. Moreover, the decision chromosomes would each have eighty genes. There is always a one to one mapping from a fuzzy state to a particular gene in the chromosome. Therefore, the locus of a gene refers to a particular fuzzy state. Genes on the Reserve Price decision chromosome only have phenotypic effect (or gene expression) if, for a particular auction state, the product of the membership values for 'expected bidder valuation', 'bidder valuation range' and 'expected bidder scale-down' is non zero. Since fuzzy sets overlap, an auction state may be described by more than one fuzzy state hence more than one gene may have a phenotypic effect. For Starting Price and Reserve Price decisions, the phenotypic effect is for the Seller agent to set these prices as positive integer values. How Seller agents make a decision and set their starting prices and reserve prices is explained in the following section.

Phenotype

In the simulated auctions, bidder valuation distributions are randomly generated. However, the Seller agent's beliefs are assumed correct i.e. what the agent assumes for the bidder valuations is correct. The auction simulator generates bidder valuations in two stages. Firstly, random values for 'expect bidder valuation'; 'bidder valuation range' and 'expected bidder scale-down' are generated between the minima and maxima set by the user. Assume these values are v, r, and s, respectively. Secondly, the auction simulator uses these values to form the Uniform distribution of bidder valuations. The Uniform distribution for bids being $U[v-r/2, v+r/2]$ and for scale-downs being $U[s/2, 3s/2]$. Although, the range of bidder scale-downs is always set to s.

However, assume the auction is Vickrey-PV. Therefore, for this auction the minimum and maximum bidder valuations are given by:

$$minVal = v\text{-}r/2 \text{ and } maxVal = v+r/2 \qquad (2)$$

A Seller agent sets its starting price and reserve prices as some integer in the interval *[minVal, maxVal]*. However, each gene within a Reserve Price decision chromosome can take positive integer values (alleles)[2] bounded by n, where n is a positive integer. The phenotypic effect of a gene having integer value p is to set the reserve price to

$$minVal + p*(maxVal\text{-}minVal)/n \qquad (3)$$

Therefore, if a gene has a value of zero then the reserve price is set to *minVal* and if the gene has value n the starting price is set to *maxVal*. Enabling users to determine the alleles of a gene allows them to make their Seller agent's strategy spaces finer or coarser. As with fuzzy set partitioning, it reduces the strategy space and makes the task of evolving good strategies easier.

For example, assume five fuzzy sets are used to partition 'expected bidder valuation' and three fuzzy sets are used to partition 'bidder valuation range'. This implies fifteen fuzzy states partition the auction state space. Also, assume that genes can take integer values from the interval *[0,6]*. Therefore, the Reserve Price decision chromosome could look like *3-0-4-2-4-5-6-1-2-6-2-2-2-4-3*. The first gene has phenotypic effect when the {very low} fuzzy set covering 'expected bidder valuation' has a non-zero membership value and the {low}[3] fuzzy set covering 'bidder valuation range' has a non-zero membership value. The second gene has phenotypic effect when the {very low} fuzzy set covering 'expected bidder valuation' has a non-zero membership value and the {medium} fuzzy set covering 'bidder valuation range' has a non-zero membership value. The third gene has a phenotypic effect when the {very low} fuzzy set covering 'expected bidder valuation' has a non-zero membership value and the {high} fuzzy set covering 'bidder valuation range' has a non-zero membership value. The fourth gene has a phenotypic effect when the {low} fuzzy set covering 'expected bidder valuation' has a non-zero membership value and the {low} fuzzy set covering 'bidder valuation range' has a non-zero membership value. And so on until the fifteenth gene that has phenotypic effect when the {very high} fuzzy set covering 'expected bidder valuation' has a non-zero membership value and the {high} fuzzy set covering 'bidder valuation range' has a non-zero membership value. The fuzzy set descriptions are subjective and not important. The essential point is the ordering of fuzzy states and the mapping

2 The user can set the alleles of a gene for the chromosomes.
3 The fuzzy sets {low}, {medium} and {high} etc. used to cover 'expected bidder valuation' and the fuzzy sets {low}, {medium} and {high} used to cover 'bidder valuation range' are different sets.

of fuzzy states to genes. Let the chromosome given above be stored in an array with 15 elements *(C[15])*.

As described earlier, a Seller agent's belief of 'expected bidder valuation' can be stored in an array of five elements, where the first element is the membership level of 'expected bidder valuation' in the fuzzy set {very low}. Element 2, is the membership level for 'expected bidder valuation' in the next fuzzy set {low}. Similarly, for elements 3, 4 and 5. In addition, 'bidder valuation range' can be stored in array but of three elements. The first element is the membership level of 'bidder valuation range' for the fuzzy set {low} and so on for the other two elements. These two arrays are combined to give a fifteen-element array, *F[15]* that summarises the auction state. If *V[i]* = (0, 0, 0.6, 0.4, 0) is the array for 'expected bidder valuation' and *R[j]* = (0, 0.2, 0.8) is the array for 'bidder valuation range' then the fifteen element array is formed using the formula (1) from earlier = *(0, 0, 0, 0, 0, 0, 0, 0.12, 0.48, 0, 0.08, 0.32, 0, 0, 0)*. So each element of *F[15]* corresponds to a unique fuzzy set combination of {expected bidder valuation and bidder valuation range}.

The Seller agent makes its decision in the following way. Firstly, *F[15]* is multiplied by *C[15]* element by element. The elements are then added together to give a total. Since membership levels are values in the interval *[0,1]* and the gene values have integer values in the interval *[0,6]* (in this example) the total value must also lie in the interval *[0,6]*. Call this total value, *t*. Therefore, t

*=(0, 0, 0, 0, 0, 0, 0, 0.12, 0.48, 0, 0.08, 0.32, 0, 0, 0) * (3, 0,4, 2, 4, 5, 6, 1, 2, 6, 2, 2, 2, 4, 3)=1.88*

Now assume the Seller agent believes the 'expected bidder valuation' to be 100 and 'bidder valuation range' to be 50. Therefore, using equations (2) and (3) from earlier,

Minimum bidder valuation = (100 – 50/2) = 75
Maximum bidder valuation = (100 + 50/2) = 125
Reserve price = *75 + t*(125-75)/6 = 91* to the nearest integer.

The procedures are the same for CV auctions except 'expected bidder scale-down' is taken into account. This reduces the valuations, e.g. minimum valuation becomes *(minVal – 3s/2)*, where *3s/2* is maximum scale-down and maximum valuation becomes *(maxVal- s/2)*, where *s/2* is minimum scale-down. Also, for English and Dutch auctions, the starting price decisions are made in exactly the same way though separate chromosomes are used for each decision.

Evolving Selling Strategies

Users can configure their auction simulators and their FGA(s) to evolve selling strategies using various parameters. So far, the auction environment, fuzzy partition and gene alleles parameters have been discussed. The remaining group of parameters used to configure the FGA consists of:

- Number of Generations used to evolve a population of Seller agent strategies;
- Number of Seller agents used in an evolving population;
- Number of bidders at the auctions;
- Number of auctions played by each Seller agent per round;
- Crossover probability and Mutation probability;
- Seller agent's initial credit.

Once these parameters have been set, the FGA is ready to run. Each Seller agent in the population takes turns to play their round of auctions. For each auction, values for 'expected bidder valuation' and 'bidder valuation range' are randomly selected between the minima and maxima set by the users. Therefore, Seller agents' strategies are tested and evolved for auctions with different bidder valuation assumptions.

Fitness	Private Value and Common Value Auctions
FPSB	$F = H - V$ for $H >= RP$ where, H is the highest bid $F = 0$ for $H < RP$
Vickrey	$F = P - V$ for $H >= RP$ where, H is highest, P is second highest bid $F = 0$ for $H < RP$
English	$F = H - V$ for $H>=RP$ & $H>=SP$ where, H is the highest bid $F = 0$ for $H<RP$ or $H < SP$
Dutch	$F = H - V$ for $SP>=H$ & $RP<=H$ where, H is the highest bid $F = 0$ for $H<RP$
	Bidder valuations ~ U[a,b] where a is the minimum valuation, b is the maximum valuation and a<=H<=b & a<=SP<=b & a<=RP<=b F is the fitness (profit or loss) H is the highest bid submitted and P is the bid price paid SP is Starting Price and RP is Reserve Price V is the value of the item: for PV auctions it is equal to the Seller's private value whilst for CV auctions it equals 'expected bidder valuation'.

Table 6.2 Fitness equations for Seller agents in simulated auctions

After each Seller agent in a population has played their round of auctions, a new generation is formed. However, the first generation is generated randomly and subsequent generations are combined by ranking the Seller agents, pairing and forming a new population, see Section 4.4.2. The Seller agents are ranked according to how much profit or loss they made in a round of auctions. This is called Seller agent's fitness. In table 6.2, the fitness functions for the reserve and starting price strategies for the eight auction method/types are shown.

A graphical example is given for the fitness function in FPSB-PV auctions, see figure 6.5. For PV auctions, the Seller agent knows the exact value of the item, i.e. private value. This parameter should be set by the user when configuring the auction environment. Users may specify any private value between the minimum and maximum bidder valuation. The Seller agent's private value applies to all the auctions regardless of the bidder valuations that are randomly generated.

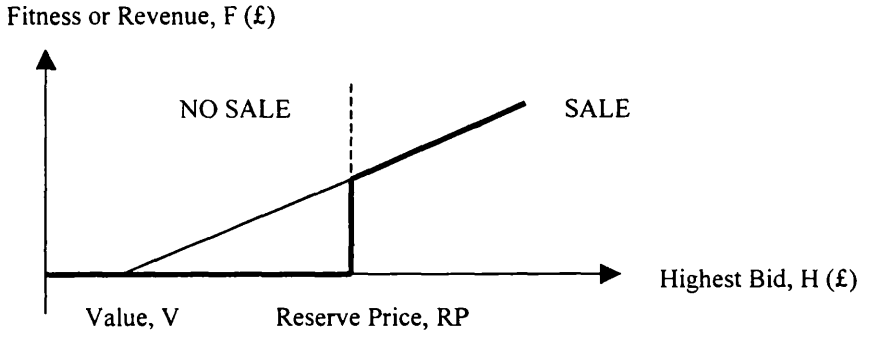


Figure 6.5 Fitness graph for Seller agents in FPSB-PV auctions

For CV-auctions, the Seller agent must estimate the value. In this simulation, the value is set to the 'expected bidder valuation' for each auction. This changes for every auction since the distribution of bidder valuations changes randomly.

Summary

This section explained how users used the auction simulator to evolve their Seller agents' selling strategies (starting prices and reserve prices) for a variety of auction types and methods. The simulated Seller agents used selected pieces of information (bidder valuations) to make strategic decisions. In particular, fuzzy logic was used to summarise this information and FGA(s) were used to encode and evolve Seller agents' strategies in various auction environments. These methods are evaluated using various test sets in Chapter 8.

6.2.2 Configuration

Users that have evolved their Seller agent strategies are ready to create the rest of the Seller agent. Although, some details have already been implicitly defined by evolving strategies, e.g. auction method and type, users still need to set the following details to completely configure a particular auction:

- Item to auction;
- Expected bidder valuation;
- Bidder valuation range;
- Expected bidder scale-down;
- Increment or decrement for asking bid in English and Dutch auctions;
- Auction start date and time.

These (together with the Seller agent's genotype and the fuzzy partitions used) are automatically marked up to form a Seller agent script. The Seller agent script is based on Standardised Generalised Mark-up Language (SGML). As described in Section 4.1.2, SGML[4] is a meta-language used to create other languages like HTML that enable structured documents to be written. Though agents are not documents, SGML could be used to formalise the structure of an agent to make it standard, verifiable, and portable.

Seller Agent Mark-up Language

The mark-up language used to mark-up Seller agents is called Seller Agent Mark-up Language, (SAML), see figure 6.6 for an example of a Seller agent script just after the auction. However, an SGML-based agent is made up of the SGML declaration, the SAML Document Type Definition (DTD), and the Seller agent script written in SAML. The SGML declaration contains all the meta-information on the tags, delimiters, and character sets used in the DTD. The DTD encodes all the rules that completely specify Seller agent scripts. These rules are made up of elements, attributes, and entities. They define the Seller agent's structure lexically, syntactically, and semantically. In other words, they specify what tags are required in the Seller agent script, in what order the tags should come and to some extent what data values can be marked up by each tag.

---

[4] In the last couple of years, the World Wide Web Consortium has developed XML, a meta-language like SGML but much simpler that enables mark-up languages to be developed. Many new mark up languages are being based on XML or changed from SGML to XML formats, e.g. Chemical Mark-up Language and Mathematical Mark-up Language, see references [BOS97], [MAT98] and [XML].

```
<SAML>
    <ID>
      <DATE TIME>03/23/98 09:17:21</DATE TIME>
      <ADDRESS>London</ADDRESS>
      <CREATOR>sotherbots</CREATOR>
    </ID>

    <AUCTION ASSUMPTIONS>
      <AUCTION METHOD>ENGLISH</AUCTION METHOD>
      <AUCTION TYPE>PV</AUCTION TYPE>
      <DELTABID>10</DELTABID>
      <EXPECTED BIDDER SCALE-DOWN>667</EXPECTED BIDDER SCALE-DOWN>
      <EXPECTED BIDDER VALUATION>3400</EXPECTED BIDDER VALUATION>
      <BIDDER VALUATION RANGE>890</BIDDER VALUATION RANGE>
      <START>02/23/98 09:20:58</START>
      <LOT ID>vintage wine</LOT ID>
    </AUCTION ASSUMPTIONS>

    <FUZZY PARTITION>
      <EXPECTED VALUATION>4:2000,5000</EXPECTED VALUATION>
      <VALUATION RANGE>3:100,500</VALUATION RANGE>
      <EXPECTED SCALE-DOWN>2:140,204</EXPECTED SCALE-DOWN>
    </FUZZY PARTITION>

    <GENOTYPE>
      <CHROMO RESERVE PRICE>429839290323</CHROMO RESERVE PRICE>
      <GENE ALLELES RESERVE PRICE CHROMO>10</GENE ALLELES RESERVE PR..>
      <CHROMO STARTING PRICE>123124524101</CHROMO STARTING PRICE>
      <GENE ALLELES STARTING PRICE CHROMO>6</GENE ALLELES STARTING PR..>
    </GENOTYPE>

    <MEMORY>
      <BIDDER COUNT>3</BIDDER COUNT>
      <LOG BIDDER>
            <BIDDER ID/BID>any1@London 03/23/98 09:16:13:Y</BIDDER ID/BID>
            <BIDDER ID/BID>any2@NewYork 03/23/98 09:12:46:N</BIDDER ID/BID>
            <BIDDER ID/BID>any3@Oslo 03/23/98 09:13:28:N</BIDDER ID/BID>
      </LOG BIDDER>

      <LOW BID>0</LOW BID>
      <SECOND HIGH BID>0</SECOND HIGH BID>
      <HIGH BID>3700</HIGH BID>
      <TOP BIDDER>any2@NewYork</TOP BIDDER>
      <RESERVE PRICE>2900</RESERVE PRICE>
      <STARTING PRICE>1500</STARTING PRICE>
      <AUCTION OUTCOME>SALE: any2@NewYork 03/23/98 Bid=3700</AUCTION ...>
      <STATE>COMPLETING</STATE>
    </MEMORY>
</SAML>
```

Figure 6.6 A Seller agent script written in SAML

An instance of a Seller agent script and its SAML DTD are called an SGML declaration, see figure 6.7. The first line indicates the document type is for Seller agents. The following lines that contain tags starting with '<! ELEMENTS...>' and '<! ENTITY...>', specify the rules. Between the tags <SAML> and </SAML> lies the actual Seller agent script.

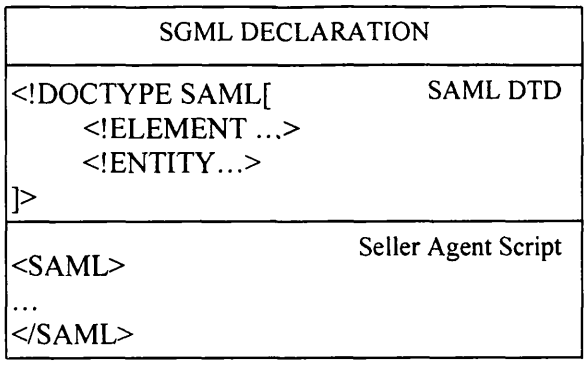


Figure 6.7 SGML-based Seller agent scripts

However, the formal SAML DTD and Backus-Naur Form (BNF) have not been written or implemented in the IAS and is left as further work. (Although, SGML DTD(s) give some indication of how agent scripts could be automatically verified using public SGML DTD(s), thereby, imposing some standard for agents much like HTML imposes a standard for web pages, see Chapter 9.3). Instead, a rudimentary 'tag checker' has been written for SAML that checks the Seller agent has the correct tags, in the right order and that they mark up the right type of data, see Appendix F.

Compulsory Tags or Lexical Parse

This section describes the type of tags required and their respective numbers that must be present in a Seller agent script. Most of the tags that mark up a Seller agent are self-explanatory. The main tags and their closing or terminating tags (represented by '</...>') must be enclosed by <SAML> and </SAML> are:

- <ID></ID>
- <AUCTION ASSUMPTIONS> </AUCTION ASSUMPTIONS>
- <FUZZY PARTITION></FUZZY PARTITION>
- <GENOTYPE></GENOTYPE>
- <MEMORY></MEMORY>

The <ID> tag contains <CREATOR>, <ADDRESS>, and <DATE TIME> tags. This uniquely specifies the agent by marking up its creator's name, web server address and the date and time that the creator created the Seller agent. The <AUCTION ASSUMPTIONS> tag marks up all the auction assumptions that are used by the Seller agent, e.g. auction method, type, and bidder valuations. All the tags are recognisable except <DELTABID>. This enables the Seller agent to control how much the auctioneer monitor increases or decreases asking bids in English and Dutch auctions. The <FUZZY PARTITION> tag marks up for each piece of information

('expected bidder valuation', 'bidder valuation range' and 'expected bidder scale-down') the minimum value, maximum value and the number of fuzzy sets used to partition them. The <GENOTYPE> tag marks up the Seller agent's Starting Price and Reserve Price decision chromosomes. The tags <GENE ALLELES ... CHROMO> set what values the genes can take. For example, if it is set to five then genes can take integer values from zero to four. The last main tag is <MEMORY>. This tag marks up all information required by the Seller agent to conduct its auction. For example, all the Bidder agents who bid in the auction are logged. In this case, there may be more than one <BIDDER ID/BID> tag. Its reserve price, starting price, and various details on the bids submitted are also stored. The <STATE> tag is very important since it is used by the Seller agent interpreter to run the agent as a finite state automaton, see Section 6.3. All the tags mentioned are mandatory (unlike with some mark-up languages in which tags may be left out of a document instance). Therefore, a Seller agent's script must contain all the tags and their closing tags for it to be successfully parsed lexically.

Tag Order or Syntax Parse

This section reviews the rules governing tag order. If a Seller script follows these tag order rules then it should be successfully parsed syntactically.

- All SAML scripts must start with <SAML> and finish with </SAML> (the closing tag).

- All tags must have their closing tags scripted at some point after them, i.e. not before. For example, <ID>... </ID> is fine but </ID>... < ID> is not.

- In general, tag order is not important, e.g. the main tags <ID>...</ID>, <GENOTYPE>...</GENOTYPE> etc. may be placed in any order between <SAML> and </SAML>. However, if a tag is embedded between one tag and its closing tag, it cannot be placed outside, e.g. the <CREATOR>, <ADDRESS> and <DATE TIME> tags are embedded between <ID> and </ID>. They can be written in any order but all must still be embedded between <ID> and </ID>.

Tag Values (Semantics)

So far, the Seller agent script has been defined by what tags it must have and the order that they must appear in. The last set of rules defines the values that can be placed between the tags, i.e. semantic information. In table 6.3, the acceptable values for each tag are defined.

Meta-tags and Tags	Value
<ID>	Meta-tag
<CREATOR>	String
<ADDRESS>	String
<DATE TIME>	Java Date Time String
<AUCTION ASSUMPTIONS>[5]	Meta-tag
<EXPECTED BIDDER VALUATION>	Positive Integer
<BIDDER VALUATION RANGE>	Positive Integer
<EXPECTED BIDDER SCALE-DOWN>	Positive Integer
<AUCTION METHOD>	FPSB, Vickrey, English or Dutch
<AUCTION TYPE>	PV or CV
<DELTABID>	Positive Integer
<START>	Java Date Time String
<LOT ID>	String
<FUZZY PARTITION>	Meta-tag
<EXPECTED VALUATION> <VALUATION RANGE> <EXPECTED SCALE-DOWN>	n:x,y, where n is the number of fuzzy sets used to partition the information space, x is minimum value and y is maximum value
<GENOTYPE>	Meta-tag
<CHROMO RESERVE PRICE>	String of integer values
<GENE ALLELES RESERVE PRICE ...>	A positive integer
<CHROMO STARTING PRICE>	String of integer values
<GENE ALLELES STARTING PRICE ...>	A positive integer
<MEMORY>[6]	Meta-tag
<BIDDER ID/BID>	Agent's ID and their bid in sealed-bid auctions or bid answer in English/Dutch auctions
<BIDDER COUNT>	Positive integer
<LOW BID>	Positive integer
<SECOND HIGH BID>	Positive integer
<HIGH BID>	Positive integer
<TOP BIDDER>	Agent's ID
<RESERVE PRICE>	Positive integer
<STARTING PRICE>	Positive integer
<AUCTION OUTCOME>	SALE | NOSALE if sale Agent's ID and winning bid
<STATE>	WAITING, PREPARING, AUCTIONING, COMPLETING

Table 6.3 Acceptable tag values for Seller agent scripts

5 Note that the Seller agent's private value for the auctioned item is not marked up. This is only used in the simulations to assess fitness and evolve good strategies.

6 These are never filled in by users. When a Seller agent is formed, they are set to zero or left blank except the value for <STATE>. This is set to WAITING. As the Seller agent runs, these tag values are updated by the Seller agent interpreter. For example, after the Seller agent has calculated its reserve price its interpreter updates its <RESERVE PRICE> tag value.

As mentioned within the table, users should not edit some tag values. However, if a user uses the GUI(s) then the SAML is generated automatically. If the user writes an SAML script and attempts to load it into the IAS, the SAML 'tag checker' can be used to parse the script[7]. Therefore, users do not have to concern themselves with SAML just as web browser users do not have to know about HTML. Primarily, SAML is used to formalise the Seller agent structure so that it can be standardised and verified within the IAS. At this point users can evolve strategies for their Seller agents, configure their Seller agents, and automatically generate their SAML scripts; the next section describes how the Seller agent automates the auction.

6.2.3 Automation

This section describes the Seller agent as a Finite State Automaton. (For details of how agent scripts are interpreted as automata, see pseudo code in Section 4.1.2 and partial source code in Appendix G). Initially the newly created SAML script is passed to the (Seller agent interpreter) object's constructor as a string. Then all its tag values required by the interpreter to run the agent as automaton and automate the selling part of the auction process are read into local variables, e.g. the <STATE> tag marks up the Seller agent's current state. Next, an auctioneer monitor object is created. This is used to mediate all communications between the Seller agent and all Bidder agents who register to join the Seller agent's auction. In particular, Seller agents need to know which Bidder agents have registered to join the auction, what their bids are etc.? Bidder agents need to know when the auction starts. What the asking bid is? And, what the outcome was? These processes are represented as a state transition diagram, see figure 6.8. The Seller agent's starting state is WAITING, its mid-states are PREPARING and AUCTIONING, and its terminating state is COMPLETING. Arrows indicate possible state transitions. The Seller events that cause state transitions and the actions they perform during in a state are explained below.

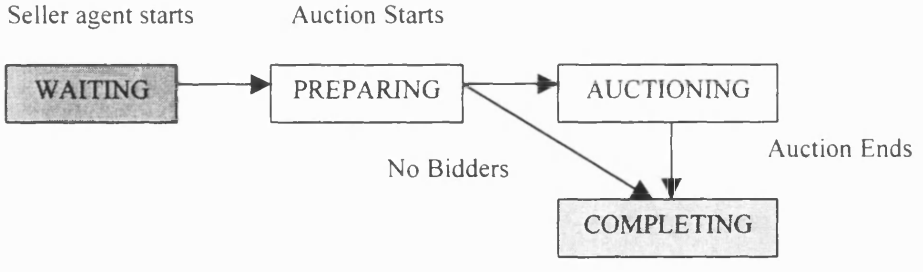


Figure 6.8 State transition diagram for the Seller agent

[7] The SAML parser does not check correlated tag values for consistency. Therefore, editing SAML scripts and parsing them still may cause the auction system to malfunction.

WAITING

The Seller agent starts in a WAITING state. It waits until it is time to start its auction at the date and time specified by the user. During this period, it checks with its auctioneer monitor to see if any new Bidder agents have registered to join its auction. If they have, it will create a new <BIDDER ID/BID> tag. Wherever possible, the Seller agent updates its script with the latest information. This provides a complete record that is accessible by the user after the Seller agent has finished. The Seller agent also writes log entries to a local log file (see Section 6.3) and updates its user's GUI with its current state and its activities, e.g. during the auction. Just before the auction starts, the interpreter updates the Seller agent's <STATE> tag value with PREPARING.

PREPARING

The Seller agent's state is now PREPARING. The interpreter knows this because it runs as a thread continually checking the Seller agent's state. The first action the Seller agent performs is to calculate its reserve price and if need be its starting price. The way it does this is exactly the same way as the simulated Seller agents calculated theirs. Firstly, it uses its fuzzy set membership levels to describe the state of the auction, i.e. what it believes the 'expected bidder valuation', 'bidder valuation range' and if need be the 'expected bidder scale-down' to be. These are held in the <FUZZY PARTITION> and <AUCTION ASSUMPTIONS> sub-tag values. Secondly, it uses its decision chromosome together with the arrays of membership levels to generate a value between zero and the maximum value for a gene allele (specified by <GENE ALLELES ...> tags. Together these values are used to calculate the reserve price and starting price. A scaling rule is used where a gene allele equal to zero implies the lowest value (minimum valuation) for the reserve price and the maximum value for a gene allele implies highest value (maximum valuation) for the reserve price.

Once the reserve price and starting price have been calculated, they are written to one of the auctioneer monitor's shared-variables. This signifies that the auction has started. The Seller agent does three additional things. Firstly, it updates its own script with reserve price and starting price. Secondly, it writes a log entry to a log file. Thirdly, it updates its user's GUI so that the user can see what is happening. Finally, it checks to see if any Bidder agents registered to participate in the auction by reading from a shared-variable in the auctioneer monitor. If no Bidder agents have registered, it updates its <STATE> tag value with COMPLETING because there cannot be an auction. Otherwise, it updates its <STATE> tag value with AUCTIONING.

AUCTIONING

What the Seller agent does next depends on the auction method. If the auction method is FPSB or Vickrey then there is no starting price. Once the Bidder agents know the auction has started, they submit their bids to the auctioneer monitor by writing to its shared-variable array. When all the Bidder agents have done this, the Seller agent can read the bids from the array. It then checks for the highest bid, second highest bid, and lowest bid. These values are used to update the relevant tag values and log entries. The Seller agent's state then changes to COMPLETING.

However, if the auction method is English or Dutch, the process is more complex. Initially, the asking bid is set to the starting price. Once the asking bid shared-variable has been set, Bidder agents read it and decide what to do (details on this are described in Section 7.2.3). However, for English auctions, the Bidder agents decide whether to accept the asking bid or not. If a Bidder agent does, it sets a shared-variable in the auctioneer monitor to 'Y' otherwise; it sets the variable to 'N'. Once all the Bidder agents have done this, the Seller agent checks to see which Bidder agents have accepted. If more than one has, the Seller agent increases the asking bid (as specified by the <DELTABID> tag value) and the whole process repeats until one (or none) Bidder agent accepts. The remaining Bidder agent is the winner. If there is a tie (i.e. no Bidder agents accept at a particular asking bid) then the Bidder agent who registered first wins. In the Dutch auction, the process is similar. Initially, the asking price is set to the starting price. Bidder agents may accept this or not by setting the appropriate shared-variables to 'Y' or 'N'. If no Bidder agent accepts then the asking bid is reduced (as specified by the <DELTABID> tag value). This process continues until the first Bidder agent accepts. The process is fair because Bidder agents replies are checked in the order of their registration i.e. first come first served. Once the auction has finished, the Seller agent's state changes to COMPLETING.

COMPLETING

The Seller agent has all the information it needs to determine the winning Bidder agent. For all the auction methods, if the highest bid is less than the Seller agent's reserve price then there is no sale. If the highest bid is greater than or equal to the reserve price then the highest bidder wins and pays its bid except in the Vickrey auction where it pays the second highest bid. All the Bidder agents are informed of the auction outcome (no-sale or sale to which bidder and its bid) by reading the outcome shared-variable. Once the Seller agent has set the auction outcome shared-variable, its interpreter thread terminates. The Seller agent script is then held as a text file within the IAS. The user may view it and keep it as a record of its automated auction.

6.3 Agent Utilities

This section briefly describes a few additional features that do not follow the normal cycle of agent creation/running, see Section 3.2.2 for details. These features include log files, agent utilities and other ways that Seller agents can be loaded into the IAS and run.

Log Files

As the Seller moves from state to state, it creates log entries. These form a complete picture of what the Seller agent did and what happened in the auctions. However, in the IAS, log entries are text strings summarising particular events. Log entries take the form: *(SellerID BidderID TimeStep EventCode Arg1 Arg2 Arg3)*. Where all the pieces of information are written as integer codes. If an event refers just to a Seller agent then *BidderID* would be represented by zero and vice versa. For example, a Seller agent (*SellerID 3*) setting its reserve price to 120 (*EventCode* 12) at *Timestep* 212 would be written as (3, 0, 212, 12, 120, 0, 0). If log entries were written in this way for all the agents and the auctions took seconds to run then they could be loaded into the animated auction player and replayed. Though this has not been implemented in the IAS, it would provide a good way show graphically what the agents are doing in the auctions.

Utilities

The remaining utilities enable users to manage their Seller agents. Users may load, run, edit and save Seller agent scripts and store them in local directories or on web servers. They may load their Seller agent scripts into their IAS system from local directories or web servers. They can then edit their Seller agent scripts and run them. Conversely, once a Seller agent has been created or has completed (no associated interpreter thread is running), users may save their agents to a local directory. In addition, they may view Seller agents' log files at any time.

Reusing Agents

Most users would create new Seller (or Bidder) agents using, firstly, an auction simulator to evolve strategies and, secondly, GUI forms to configure their agents letting their system generate the script. However, experienced users may want to edit agent scripts so that the agent's strategies can be reused. They may want to reuse agents that performed well in their auctions. Users may do this by saving their agent scripts in a local directory and when necessary reloading, editing, parsing and running them again. Reusing agents is an efficient way to utilise good agent strategies especially if they have taken a long time to evolve in the auction simulator.

Chapter 7 – The Bidder Agent

This chapter completes the description of the Internet Auction System (IAS). It describes how users search for remote auctions held by Seller agents and bid in them using automated Bidder agents. In particular, the first section describes how users evolve Bidder agent's strategies in their auction simulators. The second section describes how users use the SGML-based mark up language (Bidder Mark up Language, BAML) to script their Bidder agents. The third section describes how users run their Bidder agents so that their BAML scripts are interpreted as finite state automata. However, many of the methods used for Bidder agents are identical to the ones used for Seller agents. Therefore, to avoid repeating, only the differences are described.

7.1 Overview

The following diagram (figure 7.1) symbolically shows the various components that make up a Bidder agent. These are explained throughout this chapter. The structure is similar to the Seller agent except that the Bidder agent is mobile and requires a schedule of remote auctions sites (or user addresses) to visit. However, they are briefly introduced to give some indication of what is implemented in a Bidder agent to automate the buying process.

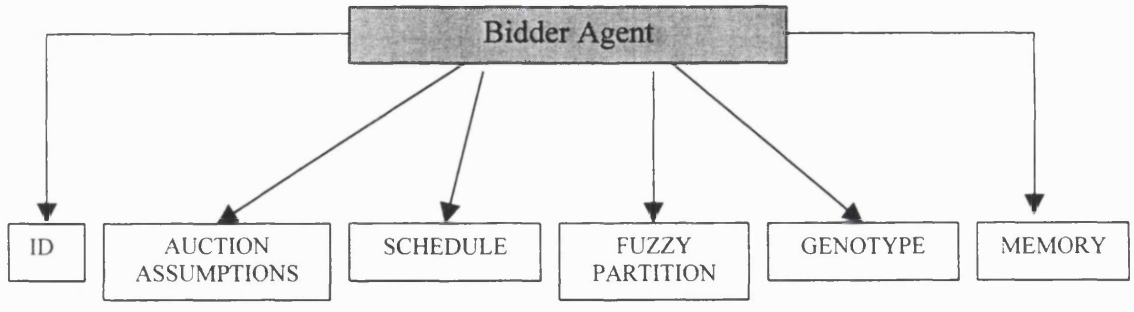


Figure 7.1 The Bidder agent

A Bidder agent has six structured components. Its first component is ID. Every Bidder agent has a unique ID that distinguishes it from all other Bidder agents in the IAS. A Bidder agent ID is made up of its user's (creator's) name, address, and the date and time it was created. Its second component is AUCTION ASSUMPTIONS. When users create Bidder agents, they supply various pieces of information, e.g. item to buy and auction method. These are used by the Bidder agent to find the auction selling the right item using the right method. Its third component is SCHEDULE. This contains all the information relating to where the Bidder agent searches for

auctions, what it finds there, and how long it should search. Its fourth component is FUZZY PARTITION. This stores information on how the Bidder agent's information space is partitioned using fuzzy sets. Its fifth component is GENOTYPE. This stores the Bidder agent's strategies for bidding and scaling down its bid in the form of genetic structures called chromosomes and genes. Its sixth component is MEMORY. This stores all the information that is gained by the Bidder agent once it starts running and visits auctions. This includes where the auction is held, what it bids at the auction, and who wins the auction? In addition, the Bidder agent's current state is stored in its memory. These components are described in the context of how users utilise their Bidder agents and how they are implemented.

7.2 Utilisation and Implementation

The process of creating Bidder agents is the same as for Seller agents. Users can create Bidder agents using their IAS GUI(s), load them from local file directories or from web servers, or reuse them. Assuming users create new Bidder agents to automate the process of searching for and bidding in remote Seller agent auctions, they create their Bidder agents in three stages, see figure 6.2 in Chapter 6. In the first stage, users evolve their Bidder agents' bidding strategies in an auction simulator. In the second stage, users enter using GUI(s) details about the auctions they want their Bidder agents to visit. In the third stage, users run and monitor their Bidder agents.

7.2.1 Adaptation

The process of adapting Bidder agent strategies is identical to adapting Seller agents. This section will only describe aspects that are specific to Bidder agents. As described in the last chapter, the auction simulator was used to adapt bidding and selling strategies. It has three groups of parameters that need setting before Bidder agents' strategies can be evolved. These are auction assumptions, fuzzy partitions, and standard parameters associated with Genetic Algorithms (GA). Parameters specific to Bidder agents are described in terms of how they are used to evolve bidding strategies within the simulated auctions. In any case, from the Bidder agent's point of view, the simulated auction is considered a game with one Seller agent and one or more bidder(s) (not actually Bidder agents) as opponents. This section gives details of what information and strategic decisions are used by Bidder agents in the simulated auctions. It also very briefly describes how Bidder agents' strategies are encoded using fuzzy sets and genetic structures. Finally, some details concerning the FGA fitness functions used to evolve Bidder agent strategies are described; for specific details on implementing the fuzzy partitioning and FGA, see Sections 4.2.2 and 6.2.

Information and Strategic Decisions

A Bidder agent's bidding strategies depends on the auction method and type [VIC61], [MCA87], [MIL82]. As explained before, the auction method determines the rules by which an item is auctioned. The four methods are FPSB, Vickrey, English, and Dutch. The auction types define how Bidder agents value the auctioned item. In a Private Value (PV) auction, all the Bidder agents know the value of the item for certain, i.e. a Bidder agent's valuations and private value is the same; though Bidder agents' private values may be different. In a Common Value (CV) auction, the item has the same value to all the Bidder agents but they may not know its precise value and have to make valuations. Bidder agents can reduce the chance of over estimating the item's value and possibly inducing a loss by scaling down their bids. Therefore, in PV auctions, the Bidder agent only needs to consider what to bid for the auctioned item. For CV auctions, the Bidder agent needs to consider what to bid and scale-down[1].

However, during real[2] (not simulated) English auctions, the Bidder agent uses the asking bid to re-evaluate their beliefs on the item's value or the other bidder valuations. There are two reasons for this. Firstly, in PV auctions the Bidder agent will want to change what it bids based on other bidders' values. Secondly, in CV auctions the Bidder agent's valuation will change depending on what others bid; this in turn affects what the agent bids (and resulting profit). Therefore, in English auctions Bidder agents use the current asking bid to decide whether to remain in the bidding. In table 7.1, a Bidder agent's strategic decisions are listed.

Strategic Decisions	FPSB	Vickrey	English	Dutch
Private Value	Bid Amount	Bid Amount	Bid Amount	Bid Amount
Common Value	Bid Amount Scale-down	Bid Amount Scale-down	Bid Amount Scale-down	Bid Amount Scale-down

Table 7.1 Bidder agent's strategic decisions

Given the auction method, type, and number of bidder opponents (set as a GA parameter), the Bidder agent must make some further assumptions about its prospective bidder opponents. To keep the modelling simple, the Bidder agent assumes that its opponents are risk neutral and the only pieces of information it uses to make decisions are:

1 Whenever Bidder agent 'scale-down' is mentioned, it implies a CV auction is being modelled
2 In simulated English auctions, a Bidder agent's beliefs about other bidder valuations are correct anyway. However, in real IAS English auctions, it would update its beliefs using the current asking-bid.

- Expected Bidder Valuation;
- Bidder Valuation Range (maximum valuation – minimum valuation);
- Expected Bidder Scale-down (for CV auctions only).

Similarly, evolving bidding strategies assumes bidder valuation distributions are Uniform. However, there are some differences. Firstly, since auctions (especially the sealed-bid and Dutch) give away little information on the other bidders' valuations; they really represent an agent's beliefs. These beliefs may or may not be accurate. In the simulation, Bidder agent's beliefs are correct, i.e. the distribution of bidders' valuations is known to the Bidder agent. This should enable the Bidder agent to evolve good strategies given correct beliefs. If this were not the case then the Bidder agent could evolve strategies using incorrect information. The evolved strategies would then be incorrectly used in the real IAS auctions. Secondly, if the Bidder agent is going to be useful in real IAS auctions it must have a complete set of strategies for expected bidder valuation distributions. In real IAS auctions, Bidder agents' beliefs may be inaccurate. However, in sealed-bid auctions Bidder agents are required to decide their bids at the start of the auction. In Dutch auctions, Bidder agents can only re-estimate at the end of the auction when it is too late. Therefore, they do not have a chance to re-estimate. However, in English auctions, Bidder agents can use the current asking bid and the fact that the auction has not ended to update their beliefs. This is described in the latter section, Phenotype. In any case, Bidder agents have a set of strategies for various auctions with different bidder valuation distributions. As with Seller agents, this enables Bidder agents to be reused in more than one auction of the same type and method. So it has strategies for all auctions of a particular method and type where:

- Expected bidder valuation lies between a minimum and maximum;
- Bidder valuation range lies between a minimum and maximum;
- Expected bidder scale-down lies between a minimum and maximum (CV auctions).

Similarly, to evolve good strategies the Bidder agent needs to participate in many auctions. However, it is assumed that in any one auction only the evolving Bidder agent has evolving strategies, all the other bidder opponents and seller have fixed strategies. Moreover, the Bidder agent opponents' valuations are randomly chosen from Uniform distributions for each new auction. However, the seller strategies are always the same. The reserve price is set to zero to make sure there is always a sale. The starting prices in English and Dutch auctions are set to minimum and maximum bidder valuations respectively. This enables all the Bidder agents to try out their strategies in many auctions environments where 'expected bidder valuation', 'bidder valuation range' and 'expected bidder scale-down' change.

Fuzzy Partitioning Auction State Spaces

The Bidder agent, like the Seller agent uses the same three pieces of information to decide what to bid and scale-down given the auction type and method. These three pieces of information are 'expected bidder valuation', 'bidder valuation range' and 'expected bidder scale-down'. The state of the auction is defined by these three pieces of information. To reduce the state space for the bidder valuation distributions, fuzzy sets are used to partition it. Since Bidder agents use the same information space as Seller agents, partitioning the state space with fuzzy sets is identical and can be reviewed in section 6.2.

Decision Chromosomes

Encoding Bidder agent strategies is identical to encoding Seller agent strategies. The information used by both agents in the simulated auctions is the same. However, in real IAS auctions, the prior beliefs given to the agents by their users are likely to differ to what the bidder valuations really are. In real IAS English auctions, the asking bid changes and the Bidder agents may update their beliefs on bidder valuations, i.e. priors to posteriors. They can then re-estimate what they want to bid. Therefore, in real English auctions Bidder agents[3] update their beliefs every time the auctioneer sets a new asking price. However, for the purposes of the simulation, it is assumed that evolving Bidder agents' beliefs are correct and they do not re-estimate bidder valuations.

In any case, for CV auctions, a Bidder agent has to decide how much to bid and scale-down. In PV auctions, it only decides how much to bid. Therefore, a Bidder agent has one chromosome for the bid decision and one for the scale-down decision. Each decision chromosome consists of a number of genes and each gene maps to a particular fuzzy state that partitions the information space (auction state space). Therefore, if there were 100 fuzzy states that are used to partition the auction state space then the decision chromosomes would have 100 genes. There is always a one to one mapping from a fuzzy state to a particular gene in the chromosome. Therefore, the locus of a gene refers to a particular fuzzy state. Genes on the Bid decision chromosome only have phenotypic effect if for a particular auction state the product of the membership values for 'expected bidder valuation', 'bidder valuation range' and 'expected scale-down' is non zero. Since fuzzy sets overlap, an auction state may be described by more than one fuzzy state hence more than one gene may have phenotypic effect. In the case of bid and scale-down decisions, the phenotypic effect is for the Bidder agent to set these as positive integer values. This is now explained in the following section.

---

3 Seller agents do not use asking bids to update their beliefs and reset reserve prices.

Phenotype

The Bidder agent makes its decisions the same way as the Seller agent. The auction simulator tests Bidder agents under different auction assumptions. Every time agents participate in an auction, the auction assumes a different 'expected bidder valuation', 'bidder valuation range', and 'expected bidder scale-down'. This will give the agents a chance to be tested in auctions where the bidder valuation distributions vary. For example, some auctions may have Bidder agents with large 'expected valuations' and 'low ranges'. In others, bidders may have 'low expected valuations' but 'high ranges'.

Assume v is 'expected bidder valuation', r is 'bidder valuation range' and s is 'expected bidder scale-down' that all lie between minima and maxima specified by the user. The auction simulator uses these values to form Uniform distributions for the bidder valuations. The Uniform distribution for valuations being $U[v-r/2, v+r/2]$ and for scale-downs being $U[s/2, 3s/2]$.

Once the bidder valuations have been randomly generated, the Bidder agent can use this information with the knowledge that it is correct. However, the problem remains as to what to bid given this information. Since $v+r/2$ *and* $v-r/2$ are the highest and lowest valuations that can be made by any bidder, the evolving Bidder agent should choose its bid between $[v-r/2, v+r/2]$. Therefore, each gene can take positive integer alleles (values) bounded by n, where n is a positive integer[4]. The phenotypic effect of a gene having integer value p such that $0<=p<=n$ is to set the bid to

$$v-r/2 + (p*r)/n. \qquad (1)$$

Therefore, if the gene has value zero then its bid is set to the minimum valuation, $v-r/2$ and if the gene has value n its bid is set to the maximum valuation, $v+r/2$. Similarly, for the scale-down decision chromosome. The scale-down is calculated from the gene values using $s/2 + (p*s)/n$, i.e. minimum scale-down, plus gene value, times the scale-down range, divided by the maximum gene value (which may be different). Finally, the Bidder agent subtracts the scale-down from its bid and submits the revised bid. In real English auctions, Bidder agents would recalculate their bids every time the auctioneer sets a new asking price but only once in real Dutch and sealed-bid auctions. However, in *all* simulated auctions the simulated Bidder agents do not re-calculate their bids because their beliefs are already correct. This whole procedure is the same for PV auctions except 'expected bidder scale-down' is not taken into account. In other words, the Bidder agent's scale-down is always set to zero.

---

4 This can be set by the user for each chromosome.

Evolving Bidding Strategies

The auction simulator and its Fuzzy-Genetic Algorithm (FGA) use the same parameters to evolve Bidder and Seller agent strategies, see Sections 5.2.2 and 6.2. However, the Bidder agent's fitness functions are different to the Seller agent's, see table 7.2. In any case, the first Bidder agent generation is generated randomly and subsequent generations are combined by ranking the Bidder agents according to their fitness, pairing and forming a new population. Bidder agents are ranked according to how much profit or loss they made in a round of auctions. The best Bidder agents combine their genetic strategies and form a new population that should be fitter than the previous. This process continues for the remaining generations. A full analysis of the evolved Bidder agent strategies is given in Chapter 8. Various test sets are used to see what effect the auction simulator and FGA parameters had on the evolving Bidder agent strategies.

Fitness Functions

Bidder agent fitness depends on the auction method/type and the Bidder agent's own valuation. These fitness equations are summarised in table 7.2. However, a Bidder agent can only make profit if it wins the auction and the bid price is less than the item's value.

Fitness	**Private Value and Common Value Auctions**
FPSB **English** **Dutch**	$F = -B + V$ for $B>=H$ $F = 0$ for $B<H$
Vickrey	$F = -H + V$ for $B>=H$ $F = 0$ for $B<H$
	F is the fitness (profit or loss) B is the evolving Bidder agent's bid H is the highest bid submitted *not* including the evolving Bidder's bid V is the value of the item, for PV auctions it is equal to the Bidder's private value whilst for CV auctions it equals expected bidder valuation Starting Price not applicable Reserve Price = 0 Bidder valuations ~ U[a,b], a is the minimum valuation, b is the maximum valuation and $a<=H<=b$ & $a<=B<=b$

Table 7.2 Fitness equations for Bidder agents in simulated auctions

As explained in Chapter 6, the Bidder agent knows the exact value of the item in PV auctions, i.e. its private value. This parameter should be set by the user when configuring the auction environment. Users may specify any private value between the minimum and maximum bidder valuation. The Bidder agent's private value applies to all the auctions, regardless of the bidder valuations that are randomly generated. For CV-auctions, the Bidder agent must estimate the value. In the simulations, the value is set to the 'expected bidder valuation' for each auction. This changes for every auction since the distribution of bidder valuations changes randomly. The following graph (figure 7.2) shows the fitness function for the bid and scale-down strategies for an English-CV auction, assuming the highest bid (excluding the evolving Bidder's bid) is less than item's value and the reserve price is ignored.

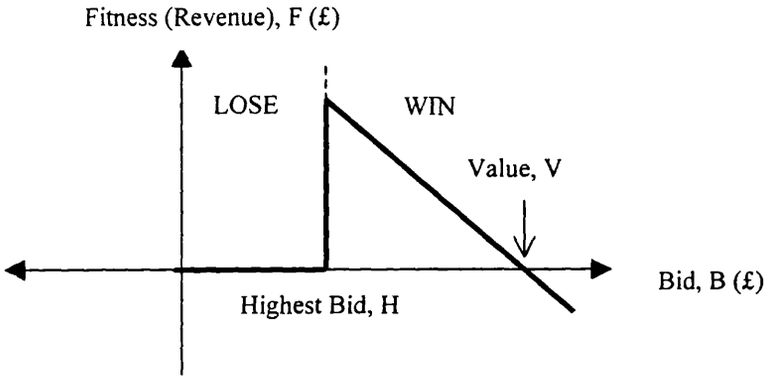


Figure 7.2 Fitness graph for Bidder agents in English-CV auction

In PV auctions, if the Bidder agent believes that its opponents value the item more highly then it cannot win the auction. In any case, it should not bid higher than its own private value. However, in (real) CV auctions, the Bidder agent does not know the item's precise value so it could bid (unwittingly) more than the item's value and make a loss. Therefore, the Bidder agent can only win an auction if its bid is the highest and can only make a profit if its bid is less than the item's value.

7.2.2 Configuration

Once a user has evolved their Bidder agent's strategies, he is ready to create the rest of the Bidder agent. Although, some details have already been implicitly defined by evolving strategies (e.g. auction method), users still need to set the following details to configure the Bidder agent:

- Item to buy;
- Expected bidder valuation;
- Bidder valuation range;

- Expected bidder scale-down.

- Schedule of auction sites to visit (see section, Schedule);
  - List of auction sites (names and address of users that may be auctioning items);
  - Bargain price (at which Bidder agent should immediately join an auction);
  - Time limit on when Bidder agent stops searching for new auctions;
  - Time limit on when Bidder agent should return home.

These details (together with the Bidder agent's genotype and fuzzy partition) are then automatically marked up in the Bidder agent script. The way the Bidder agent is marked-up is similar to how the Seller agent is marked up. However, the Bidder agent is more complex and has more features to be marked up and scripted.

Bidder Agent Mark-up Language

The mark-up language used to mark-up Bidder agents is called Bidder Agent Mark-up Language, BAML. (See figure 7.4 for an example of a Bidder agent script created by a user that has visited two auctions and decided to join the first auction but has not yet bid). The rules for BAML are derived from SGML and are called the BAML Document Type Definition (DTD). These define all possible Bidder agent scripts. Though the BAML DTD and BNF have not been implemented in the IAS, it gives an indication of how Bidder agents could be automatically verified using public BAML DTD(s). Moreover, they would impose a standard for Bidder agents, see Section 9.3. However, in the IAS, users parse their Bidder agents lexically, syntactically and semantically using simple 'tag checkers', see Appendix F for BAML 'tag checker' source code.

Compulsory Tags or Lexical Parse

Many of the tags that mark up a Bidder agent are identical to tags used to mark up Seller agents. The main tags that must be enclosed by <BAML> and </BAML> are <ID>, <AUCTION ASSUMPTIONS>, <SCHEDULE>, <FUZZY PARTITION>, <GENOTYPE> and <MEMORY>. The <ID> tag marks up the Bidder agent's identification. The auction assumptions are the same as for the Seller agent, e.g. item the Bidder agent wants to bid for, auction method and type it must bid in, its beliefs about bidder valuations. The <FUZZY PARTITION> tags are the same as for the Seller agent. The <GENOTYPE> marks up two decision chromosomes, one for the bid decision and one for the scale-down decision. The alleles of the gene are marked up using the <GENE ALLELES ...> tags. However, the main differences between Bidder agents and Seller agents are within the <SCHEDULE> and <MEMORY> tags.

```
<BAML>
    <ID>
        <DATE TIME>03/23/98 09:16:13</DATE TIME>
        <ADDRESS>NewYork</ADDRESS>
        <CREATOR>any2</CREATOR>
    </ID>

    <AUCTION ASSUMPTIONS>
        <AUCTION METHOD>ENGLISH</AUCTION METHOD>
        <AUCTION TYPE>PV</AUCTION TYPE>
        <EXPECTED BIDDER SCALE-DOWN>700</EXPECTED BIDDER SCALE-DOWN>
        <EXPECTED BIDDER VALUATION>3800</EXPECTED BIDDER VALUATION>
        <BIDDER VALUATION RANGE>800</BIDDER VALUATION RANGE>
        <LOT ID>VINTAGE WINE</LOT ID>
    </AUCTION ASSUMPTIONS>
    <SCHEDULE>
        <IA REF>
                <IA REF STATUS>CHOSEN</IA REF STATUS>
                <AUCTIONEER>Sothebots</AUCTIONEER>
                <IA ADDRESS>London</IA ADDRESS>
                <START>02/23/98 09:20:58</START>
                <APPROX PRICE>3400</APPROX PRICE>
        </IA REF>
        <IA REF>
                <IA REF STATUS>NO SUITABLE AUCTIONS</IA REF STATUS>
                <AUCTIONEER>ChristiesBots</AUCTIONEER>
                <IA ADDRESS>NewYork</IA ADDRESS>
                <START></START>
                <APPROX PRICE></APPROX PRICE>
        </IA REF>
        <IA REFS MARKER>1</IA REFS MARKER>
        <IA REFS COUNT>2</IA REFS COUNT>
        <BARGAIN PRICE>2500</BARGAIN PRICE>
        <RETURN BY>03/24/98 09:16:03</RETURN BY>
        <SEARCH UNTIL>03/24/98 09:16:03</SEARCH UNTIL>
    </SCHEDULE>

    <FUZZY PARTITION>
        <EXPECTED VALUATION>5:1500,5500</EXPECTED VALUATION>
        <VALUATION RANGE>3:400,900</VALUATION RANGE>
        <EXPECTED SCALE-DOWN>2:100,200</EXPECTED SCALE-DOWN>
    </FUZZY PARTITION>
    <GENOTYPE STRATEGIES>
        <CHROMO BID>111542154512410</CHROMO BID>
        <GENE ALLELES BID CHROMO>6</GENE ALLELES BID CHROMO>
        <CHROMO SCALE-DOWN>11010001010010100110010100 1101</CHROMO SCALE-DOWN>
        <GENE ALLELES SCALE-DOWN CHROMO>2</GENE ALLELES SCALE-DOWN CHROMO>
    </GENOTYPE STRATEGIES>

    <MEMORY>
        <AUCTION LOG>
          <LOCATION></LOCATION>
          <OUTCOME></OUTCOME>
          <STARTING PRICE>0</STARTING PRICE>
          <RESERVE PRICE>0</RESERVE PRICE>
          <BID OFFER>0</BID OFFER>
          <SCALE-DOWN>0</SCALE-DOWN>
        </AUCTION LOG>
        <AUCTIONS PLAYED>0</AUCTIONS PLAYED>
        <STATE>OBSERVING</STATE>
    </MEMORY>
</BAML>
```

Figure 7.3 A Bidder agent script written in BAML

Schedule

A Bidder agent's schedule consists of one or more *Internet auction site references.* These contain the user-specified auction addresses that the Bidder agent is required to visit. Moreover, auction references enable Bidder agents to keep track of the auctions they have visited and store some details about the auctions. An auction reference consists of the following tagged data:

```
<IA REF>
    <AUCTIONEER>ChristiesBots</AUCTIONEER>
    <IA ADDRESS>newyork</IA ADDRESS>
    <START></START>
    <APPROX PRICE></APPROX PRICE>
    <IA REF STATUS>NOT ACCEPTABLE</IA REF STATUS>
</IA REF>
```

When users specify where they want their Bidder agents to visit they list one or more usernames and addresses that they believe are holding auctions, i.e. auctioneers. The <AUCTIONEER> and <IA ADDRESS> tags mark up the user's (not Seller agent's) username and address that the Bidder agent should visit. Also, <IA REF STATUS> marks up the Bidder agent's visit status, i.e. has the Bidder agent already visited or needs to visit etc. The <START> and <APPROX PRICE> tags are only relevant if the Bidder agent finds a suitable auction (Seller agent) whilst visiting a user's auction site. A suitable auction is one that has the following criteria matched:

- Item for sale is correct;
- Auction method and type is correct;
- Start date and time is acceptable.

If a Bidder agent finds that these criteria match its own then it will check the Seller agent's estimated sale price (approximate price) for the item. The Bidder agent then modifies its own script by writing this information, together with when the auction starts in the appropriate auction reference tags. Users may specify when their Bidder agents must return home by. Moreover, they may specify when their Bidder agents should stop searching for auction sites and join one. Finally, they may specify a bargain price that indicates when a Bidder agent should stop visiting auctions and immediately join the current one. These stopping policies are marked up using <RETURN BY>, <SEARCH UNTIL>, and <BARGAIN PRICE> tags. Therefore, the schedule of auction references define a set of auction sites that Bidder agents should attempt to visit, subject to their user-defined stopping policies. However, if Bidder agents have time they may search for new auction sites. Each new auction site they visit, they create a new auction reference by embedding new <IA REF> and </IA REF> tags and their sub-tags within their own scripts. For further details on Bidder agent co-operation, see section 7.3.

Memory

Bidder and Seller agents memorise different aspects of the Internet auction process. The Seller agent only ever participates in one auction in its lifetime, but the Bidder agent may participate in many (if it keeps losing the auctions it attends). Therefore, Bidder agents mark up more data in their memory component. In particular, for every auction it bids in, it marks up the auction location (username and address), the reserve and starting prices, its bid and scale-down and the outcome of the auction. Each time it participates in an auction it will mark up an additional auction log:

```
<AUCTION LOG>
   <LOCATION>sotherbots</LOCATION>
   <OUTCOME>SALE: Bidder agent ID 01293 bid = 155</OUTCOME>
   <STARTING PRICE>100</STARTING PRICE>
   <RESERVE PRICE>130</RESERVE PRICE>
   <BID OFFER>140</BID OFFER>
   <SCALE-DOWN>0</SCALE-DOWN>
</AUCTION LOG>
```

The Bidder agent will continue to visit or search for new auctions and update its memory with new auction logs until it either wins an auction, or has to stop because of its stopping policy. The remaining two tags are <AUCTIONS PLAYED> and <STATE>. These mark up the number of auctions the Bidder agent has participated in and its current state. The Bidder agent's state is used by its interpreter to run the agent as an automaton, see section 7.3.

Tag Order and Values

Tag order rules applies identically to both Seller and Bidder agent scripts. For the rules, see Section 6.2. Similarly, a Bidder agent's tag values are described in tabular form; table 7.3 lists all the tags and their values that may appear in a Bidder agent script[5].

Users never fill in the values for the memory tags because the BAML is generated automatically. The values are set to zero or blank except the value for <STATE>, which is set to STARTING. As the Bidder agent runs, these may be updated by the Bidder agent. For example, after the Bidder agent has participated in an auction it will create an auction log and fill in all the details. If users write BAML scripts and attempt to run them, the scripts are automatically parsed with the BAML 'tag checker'. However, BAML is primarily used to formalise the Bidder agent structure so that it can be standardised and verified within the IAS. After users have evolved their Bidder agent strategies, configured and generated their BAML scripts, they can run them using local Bidder agent interpreters. This is discussed next.

Meta-Tags and Tags	Value
<ID>	Meta-tag
<CREATOR>	String
<ADDRESS>	String
< DATE TIME>	Java date time string
<AUCTION ASSUMPTIONS>	Meta-tag
<EXPECTED BIDDER VALUATION>	Positive Integer
<BIDDER VALUATION RANGE>	Positive Integer
<EXPECTED BIDDER SCALE-DOWN>	Positive Integer
<AUCTION METHOD>	FPSB, Vickrey, English or Dutch
<AUCTION TYPE>	PV or CV
<LOT ID>	String
<SCHEDULE>	Meta-tag
<IA REF>	Meta-tag
<IA REF STATUS>	TODO, REJECTED, ABANDON, DONE, NEXT, ACCEPTABLE, NOT ACCEPTABLE, CHOSEN, NOSALE, WON, LOST
<AUCTIONEER>	String
<IA ADDRESS>	String
<START>	Java date time string
<APPROX PRICE>	Positive Integer
<IA REFS MARKER>	Positive Integer
<IA REFS COUNT>	Positive Integer
<BARGAIN PRICE>	Positive Integer
<RETURN BY>	Java date time string
<SEARCH UNTIL>	Java date time string
<FUZZY PARTITION>	Meta-tag
<EXPECTED VALUATION>	n:x,y, where n is the number of fuzzy sets used to partition the information space, x is minimum value and y is maximum value
<VALUATION RANGE>	
<EXPECTED SCALE-DOWN>	
<GENOTYPE>	Meta-tag
<CHROMO BID>	String of integer values
<GENE ALLELES BID CHROMO>	Positive integer
<CHROMO SCALE-DOWN>	String of integer values
<GENE ALLELES SCALE-DOWN C...>	Positive integer
<MEMORY>	Meta-tag
<AUCTION LOG>	Meta-tag
<LOCATION>	String
<OUTCOME>	SALE | NO SALE (if sale, Bidder agent's ID and winning bid is included)
<STARTING PRICE>	Positive Integer
<RESERVE PRICE>	Positive Integer
<BID OFFER>	Positive Integer
<SCALE-DOWN>	Positive Integer
<AUCTIONS PLAYED>	Positive Integer
<STATE>	STARTING, OBSERVING, COOPERATING, BIDDING, DECIDING, REGISTERING, MOVING, COMPLETING, HOME

Table 7.3 Acceptable tag values for Bidder agent scripts

5 Note that the Bidder agent's private value for the auctioned item is not marked up. This is only used in the simulations to assess fitness and evolve good strategies.

7.2.3 Automation

This section describes how the Bidder agent is interpreted as a Finite State Automaton. The following state transition diagram (figure 7.4) shows the various state transitions that the Bidder agent goes through as it automates the buying process in the IAS. Arrows indicate possible state transitions. However, Bidder agents always start in the STARTING state and terminate in the HOME state.

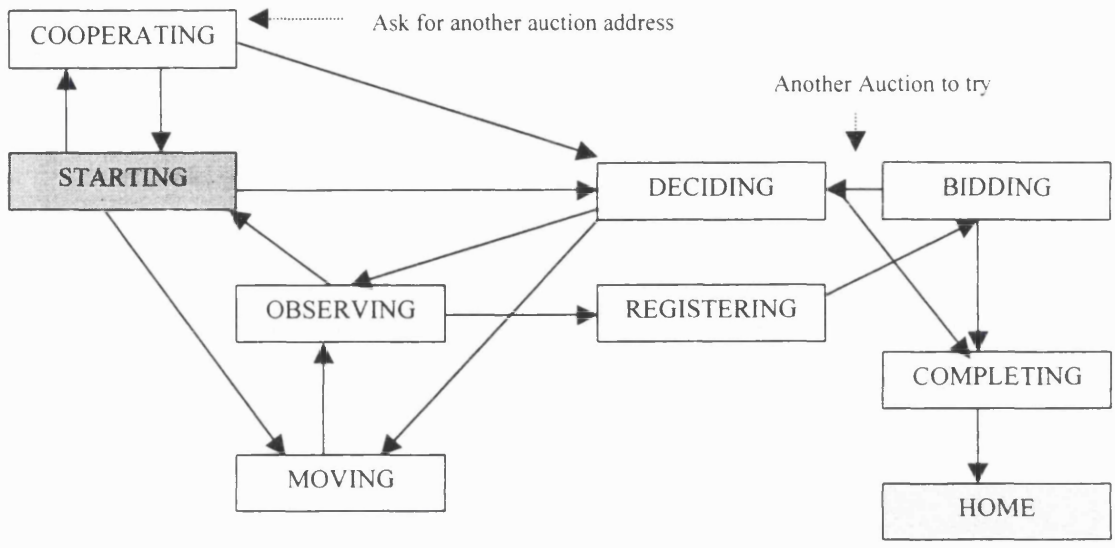


Figure 7.4 State transition diagram for the Bidder agent

Interpretation

Similarly to SAML scripts, BAML scripts are passed to (Bidder agent interpreter) object constructors as strings. Firstly, all the tag values required by the interpreter to run the agent as an automaton are read into local variables. Bidder agents start in the STARTING state. Its interpreter then checks for events, performs actions on its behalf and if need be changes the Bidder agent's state. The way the interpreter works is simplified by the following pseudo code, also see the partial source code in Appendix G.

```
for(;;){
        if      state = STARTING     do {this}
        else if state = MOVING       do {that}
        else if state = OBSERVING    do {something else}
        else if state = HOME         do {stop thread}
        else ....                    do {error unknown state}
}
```

The interpreter continually loops checking the Bidder agent's state, checking events and performing actions, In addition, the interpreter changes the Bidder agent's state by updating the

<STATE> tag. Therefore when the interpreter checks the Bidder agent's state it would have already changed and will do the appropriate actions. This continues wherever the Bidder agent is within the IAS because every user has identical BAML interpreters. Therefore, when Bidder agent scripts move from one computer to the next, they are reloaded into new interpreters and run locally. Eventually Bidder agents return home to the users that created them. Once they reach home, their states should change to HOME and they stop running. At this point, the interpreter thread is stopped and the Bidder agent becomes a passive BAML script. The Bidder agent is now described as it moves from a STARTING state to HOME state. For each state, the events that cause a Bidder agent to move into the state, possible state transitions and the actions they perform during the state are explained. However, during all the state transitions, Bidder agents create log entries by writing to local and remote log files. In addition, they update their users with information by posting messages back to their users.

STARTING

The Bidder agent starts by examining its schedule of auction sites to visit specified by its user. In particular, the Bidder agent checks the statuses of its auction references. Initially, all the Bidder agent's auction reference statuses would be TODO. Therefore, the Bidder agent picks the first auction reference in its schedule with the TODO status. It then attempts to move to the destination. However, before it does this, it checks the destination exists and then asks permission to move to the destination. Users holding auctions (auction sites) may refuse Bidder agents, so they need to check first (see the latter section, OBSERVING for more details). If the Bidder agent is allowed to move to the destination, it will update its own <STATE> tag to MOVING and the corresponding <IA REF> tag to NEXT. This enables the Bidder agent to keep track of which auction sites it has visited and the ones it still needs to visit.

MOVING

Once the Bidder agent's status has been changed to MOVING, it is posted to the destination via the destination's web server. The agent router routes the Bidder agent to the correct destination, see Section 3.1.2. Just before it is posted, the Bidder agent's status is updated to OBSERVING.

OBSERVING

The Bidder agent script is now located at its destination. Since every computer within the IAS can create interpreter objects, it can be run locally. The only difference is that the Bidder agent's state was STARTING when it was first run on its user's computer and now it has moved to its destination, it is in an OBSERVING state. The first action the Bidder agent does on arriving at a

new destination (except when it moves home) is to disclose all its auction references (only username and address). All Bidder agents must do this. This enables Bidder agents to share knowledge of auction sites. Sharing auction references becomes important when Bidder agents want to find new auction sites, see section on COOPERATING below. Now that the Bidder agent is in an OBSERVING state, it checks local auctions (i.e. Seller agents) by examining the Seller agents' auctioneer monitors. Remember, every time a Seller agent wanted to hold an auction it needs to create an auctioneer monitor. Since users may create as many Seller agents as they wish, Bidder agents may need to check many auctioneer monitors for the following:

- Item for sale matches the item it wants to buy;
- Auction type and method matches its own;
- Auction starts before it has to return home;
- Item's estimated sale price.

If these match the Bidder agent's own criteria then it can do one of two things. If the estimated sale price is below its own bargain price then it updates its own auction reference with various details[6] e.g. estimated sale price and starting date and time. Then it joins the local auction by registering. The Bidder agent updates its status with REGISTERING and updates the auction reference status with CHOSEN. If the Seller agent's approximate price is higher than the Bidder agent's bargain price then the Bidder agent still updates it auction reference with the details. However, the auction reference status is updated with ACCEPTABLE and the Bidder agent status is updated with STARTING. This means the Bidder agent found an acceptable auction but not a 'bargain' auction. If the Bidder agent did not find any suitable auctions, it will still update its status to STARTING but this time it will update the auction reference status with NOT ACCEPTABLE.

Now that the Bidder agent is in a STARTING state again, it checks its schedule for another auction to visit. It will pick the next auction reference that has a TODO status. Again, it will ask the destination for permission to move to it. If the destination replies then it could reply with either 'accepted' or 'rejected'. If it is 'accepted', its status is updated to MOVING and the Bidder agent does the same as before. If it is 'rejected', the Bidder agent makes a note of this by updating the auction reference status with REJECTED. Otherwise, it could be that the destination is not running or does not exist in which case the auction reference status is updated to ABANDON.

---

6 It does not need to mark up the Seller agent's ID because Seller agents can only auction one item at a time. Therefore, Seller agents cannot simultaneously sell identical items at a user's auction site.

The Bidder agent would visit and observe each auction site listed in its schedule. Eventually, it either runs out of auctions to visit or runs out of time for searching. If the Bidder agent runs out of time, it changes it status to DECIDING (see latter section). However, if the Bidder agent has time, it can co-operate with the local auction site by asking for a new auction site. Since, every Bidder agent passing through an auction site has to disclose all its known auction site addresses; the local auction site may possess a comprehensive list of addresses. It may have an auction site unknown to the Bidder agent or at least not in the Bidder agent's schedule. As more and more Bidder agents move around the IAS, the distribution of auction site addresses increases. If the Bidder agent co-operates then it changes its status to COOPERATING. The following section explains how it receives its new auction site address.

COOPERATING

The Bidder agent checks the list of auction site addresses held by the current auction site. It checks them one at a time to see if it already knows the address from its own auction references. If it finds a new one then it inserts the new auction reference into its script. It then reverts to a STARTING state, and continues the usual sequence of moving and observing. However, if it cannot find a new auction site address at the local auction site then it retraces its moves to previous auction sites and asks them for new addresses, see figure 7.5.

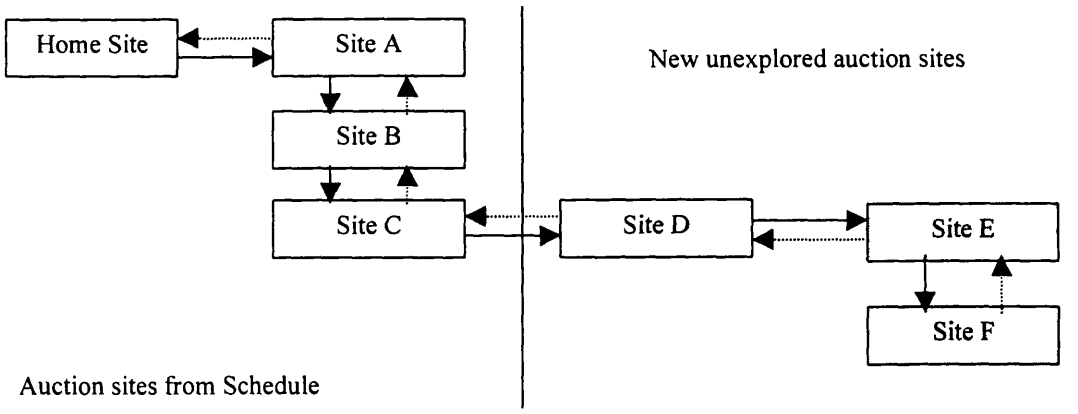


Figure 7.5 Bidder agent exploring new auction sites in the IAS

The retracing mechanism works as follows. Assume a Bidder agent has three auction sites in its schedule, call these 'A', 'B' and 'C'. Once the Bidder agent has visited all three sites, it can co-operate with the last auction site, 'C'. It will check all the auction addresses held at 'C' and on finding the first new auction site, i.e. one it has not visited, it will move to it - call this 'D'. Similarly, if the Bidder has time it will ask for a new auction at 'D' and so on until either it runs out of time or the current auction site does not have any new auction site addresses. Assume site

'D' has one additional new auction site called 'E'. The Bidder agent will visit 'E' and then asks 'E' for a new address. Assume site 'E' has one new address called 'F'. However, 'F' has no new addresses. The Bidder agent then retraces it route from 'F' to 'A' (via 'E', 'D', 'C' and 'B'). If any of these intermediary sites has a new address, the Bidder agent will again explore that branch of the IAS moving to new auction sites. If not, the Bidder agent will keep retracing until eventually returning to 'A'. It will then make a decision whether or not to register with an auction site. In this case, the Bidder agent will change its status to DECIDING.

DECIDING

If a Bidder agent has either run out of time, or run out of new auction sites to try, it then decides whether to register with one of the auctions it has visited. The decision to register at an auction is made simpler because all the auctions that the Bidder agent visited are stored in its auction references. The Bidder agent checks its auction references that have ACCEPTABLE statuses. Out of these, it picks the one that advertised the lowest estimated sale price. In addition, it will check that the auction has not already started. It then changes the auction reference status to CHOSEN. If the chosen auction site is remote it will have to move there, otherwise it remains where it is. Before registering, it changes its state back to OBSERVING. It makes some final checks that the auction is the right one and then decides to register by changing its state to REGISTER. However, if none of the auctions is suitable, it will change its state to COMPLETING and return home.

REGISTERING

A Bidder agent cannot register at an auction once it has started. Conversely, once a Bidder agent has registered to participate in an auction it cannot change its mind. It will have to wait until the auction ends before moving and checking other auctions. When a Bidder agent registers at an auction site, it looks for the Seller agent's auctioneer monitor. Once it has access to the auctioneer monitor (because they are synchronised), it adds its own ID to the auctioneer monitor's register.

An auctioneer monitor's register contains a list of all the Bidder agents registered to participate in the Seller agent's auction. When the auction eventually starts, the Seller agent will use this register to check the Bidder agent's submitted bids and conduct the auction. Once a Bidder agent has registered, it will wait until the auction starts. It continually checks the current time until the auction start time and then changes its status to BIDDING.

BIDDING

In any one auction, there will be a Seller agent; one or more Bidder agents and the mediating auctioneer monitor. Since the agents are concurrently running threads, the auctioneer monitor synchronises agents while they read and write to its shared-variables. However, synchronisation depends on the auction method.

Generally, the Seller agent initiates the auction by setting an appropriate shared-variable in the auctioneer monitor. The Bidder agents are notified (by a system call) that the Seller agent has done this. They may then read the asking bid (in English and Dutch auctions) or set their sealed-bids (in FPSB and Vickrey auctions). The cycle of the Seller agent setting the asking bid, Bidder agents reading the asking bid, Bidder agents replying and the Seller agent reading the replies are all strictly controlled by the auctioneer monitor, see section 5.1.2. However, once the auction is over, the Seller agent updates the auctioneer monitor with one of three outcomes and allows the Bidder agents to read the outcome.

Firstly, the auction could result in a NOSALE. This happens when the Seller agent's reserve price is higher than any bid. The Bidder agent will then change its status to DECIDING and its auction reference status to NOSALE (what happens next is explained below). Secondly, the Bidder agent could win the auction. In this case, the Bidder agent has completed its task. It changes its state to COMPLETING and its auction reference status to WON. Thirdly, and more likely, the Bidder agent loses the auction and is required to find and participate in another auction. In this case, it changes its state to DECIDING and updates its auction reference state to LOST. It then checks its auction references for an another 'acceptable' auction and attempts to move to that auction site and if possible register. Losing Bidder agents continue to search and participate in auctions until, either they run out of 'acceptable' auctions to try or they run out of time and have to return home. Whether a Bidder agent wins an auction or keeps losing eventually it returns home by changing it state to COMPLETING.

COMPLETING & HOME

Whether the Bidder agent has won or lost, it needs to return home to let the user examine what it did. The Bidder agent is posted home from its current auction site to its home. Before it is sent home, its state is changed to HOME. Once a Bidder agent returns home, the agent's interpreter thread is stopped and the Bidder agent no longer runs. The Bidder agent script may be examined by the user to see, where it went and what it did, from its creation to its final return.

Chapter 8 - Assessment

This chapter describes the tests and results that are used to evaluate the IAS and auction simulator. The IAS is tested in two ways. Firstly, Bidder agents are tested to see how quickly they navigate around IAS(s) with different numbers of auction sites and concurrently running Bidder agents. Secondly, the time taken for agents to automate the four auction methods are measured given different numbers of Bidder agents. The auction simulator is also tested in two ways. Firstly, the various parameters used to configure the auction simulator are analysed to see how they affect the convergence and fitness of agent strategies. Secondly, simulated Bidder and Seller agents' strategies are evolved in auctions in which their opponents have fixed strategies. Where possible these results are compared with results obtained in auction theory.

8.1 Evaluating the Internet Auction System

The effectiveness of using Adaptable Mobile Agents (AMA) to design Bidder agents and adaptable agents to design Seller agents to automate auctions within the IAS is assessed in two ways. Firstly, Bidder Agents are tested to see how long they take to navigate around IAS(s) with various numbers of auction sites and concurrently running Bidder agents. This should indicate how many agents could be used in the IAS before it becomes too slow or unstable. However, the number of concurrently running agents within an IAS is always going to be limited by the underlying computer systems, e.g. processor speed or memory limitations. Secondly, after the Bidder agents have searched the IAS for auctions, the time taken by agents to conduct the auction is assessed. How long auctions take is tested for the four auction methods with various numbers of Bidder agents bidding. These assessments are important because they should demonstrate whether the IAS is stable, efficient and an effective way of automating Internet auctions.

All the tests are carried out on one computer using a 200MHz Pentium processor and 32MB of RAM; i.e. all users' auction sites, Bidder agents, and web server run on the same computer. However, each auction site uses a different agent server. Therefore, Bidder agents still have to be routed by the web server and be sent through a socket connection to get to their auction sites. In addition, only one web server is used to route the Bidder agents and each user's auction site has at most one Seller agent running. However, the times taken by Bidder agents to move from one auction site to another would be slower if the auction sites were running on remote computers. Routing and network traffic would slow Bidder agents down in distributed IAS(s). Even so, the times taken by Bidder agents to navigate local IAS(s) represent the best times they could achieve.

8.1.1 Testing for Bidder Agent Navigation

The following four factors were varied to test how effectively Bidder agents navigated the IAS:

- Number of auction sites that the Bidder agent has in its schedule;
- Number of auction sites that are not running;
- Number of auction sites in the IAS that are unknown to it and it needs to find;
- Number of other Bidder agents within the IAS.

Bidder agents are tested to see how long they take to complete their schedules, search for new auction sites, and eventually register with the most acceptable Seller agent at one of the auction sites. This is called Bidder agent navigation duration. In all the tests, it is assumed that Bidder agents search the IAS until they have visited every auction site. They are not subject to any time constraints.

The following three graphs and table summarise the results. The first graph assumes the number of auction sites that the Bidder agent has in its schedule varies but the other three factors are kept fixed. The second graph assumes number of auction sites that are not running varies and the other three factors are kept fixed. Similarly, for the other table and graph.

Variable Number of Auction Sites in the IAS

The tests covered in figure 8.1 assumes all the auction sites within the IAS are running, i.e. none are incorrect or not running. In addition, the tests assume that only one Bidder agent is searching the IAS. These tests should show how increasing the number of auction sites affects the time taken for Bidder agents to visit them.

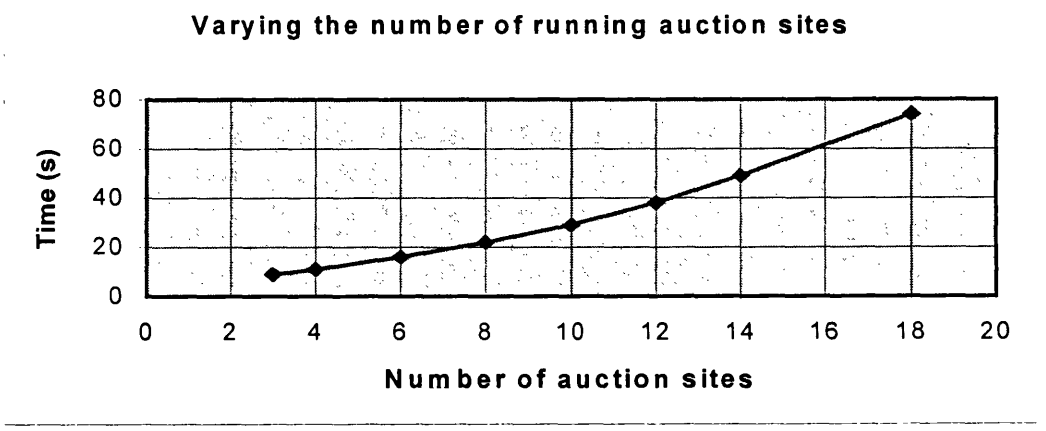


Figure 8.1 Time taken by a Bidder agent to visit all running auction sites

Discussion

The IAS was tested to see how long it took one Bidder agent to visit all the auction sites and register with the most favourable Seller agent. The maximum number of auction sites tested was 18. This appeared to be platform dependent; i.e. only eighteen listening agent server ports could be opened on one computer running Windows 95. However, on other platforms the number of auction sites is potentially much greater, e.g. UNIX or Windows NT. The results in figure 8.1 show that a Bidder agent takes less than one minute to visit up to fourteen auction sites. This is, of course, subject to network conditions. However, it shows that the Bidder agent could search auction sites in an acceptably fast time. As the number of sites increase the time taken per site increases almost linearly. For example, in most of the tests the Bidder agent takes on average 3 seconds to visit, check and leave an auction site. Only when the number of auction sites reaches 18 does it take the Bidder agent 4 seconds per auction site. These tests show that the time taken by Bidder agents to search and visit increasing numbers of auction sites within the IAS increases almost linearly at least up to 18 auction sites.

Variable Number of Non-running Auction Sites in the IAS

The tests covered in figure 8.2 assume that there are potentially 18 auction sites in the IAS. However, some of them are incorrect or not running. The Bidder agent has a complete schedule of eighteen auction sites and does not have to search for new auction sites. In addition, the tests assume that only one Bidder agent is searching the IAS. These tests should show how increasing the number of incorrect sites in the IAS affects the time taken by the Bidder agent to visit all auction sites listed in its schedule.

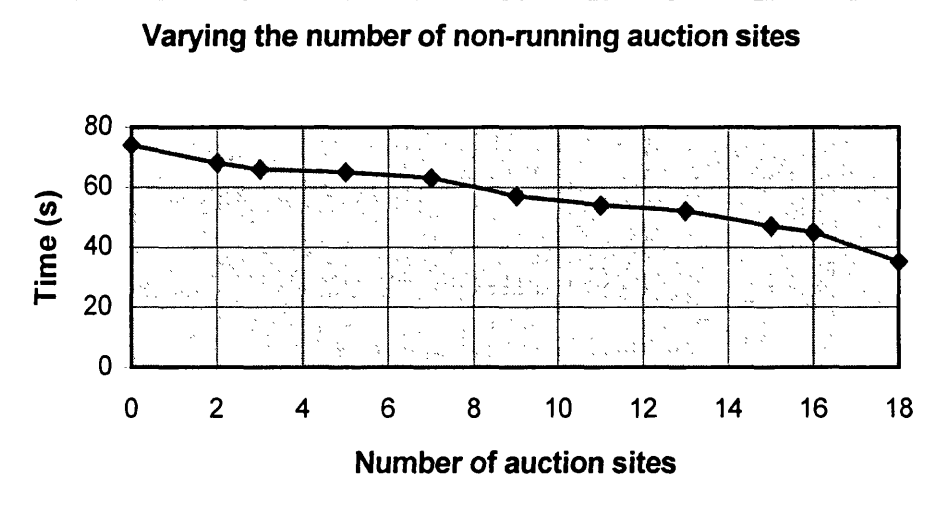


Figure 8.2 Time taken by a Bidder agent to visit all running/non-running auction sites

Discussion

The results in figure 8.2 show that as the number of incorrect auction sites increases, the Bidder agent takes less time to complete its schedule. This is because the Bidder agent checks the destination exists before it moves to it. Since, it cannot move to the auction site, it checks another auction site in its schedule and attempts to visit. Interestingly, if all the sites are incorrect, the Bidder agent still takes 35 seconds to confirm this. Therefore, Bidder agents spends almost half their time checking the destination is correct and half with moving to a destination. The tests demonstrate that Bidder agents attempting to visit auction sites that either do not exist or are not currently running do not have an adverse affect on the times taken by them to navigate the IAS.

Variable Number of Unknown Auction Sites

The tests covered in table 8.1 assume there are potentially ten auction sites in the IAS. However, a certain number are unknown to the Bidder agent, i.e. the Bidder agent has to ask its current auction site to obtain the new auction site address. The Bidder agent has a partial schedule of ten auction sites and all of them are correct and running. In addition, these tests assume that only one Bidder agent is searching the IAS. To implement these tests, initially one Bidder agent is created so that it visits the ten auction sites. As it does so, it discloses the ten auction sites addresses to ...ction site it visits. Another Bidder agent is created with various partial schedules, i.e. with one or more of the auction sites. Firstly, the Bidder agent visits the auction sites listed in its schedule. Then it asks for new (unknown to itself) auction site addresses. It visits new auction sites until there are no new sites to visit. These tests should show how increasing the number of unknown auction sites affects the time taken by Bidder agents to obtain the new auction sites addresses in the IAS.

Unknown Sites	**0**[1]	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Known Sites	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**
Time (m/s)	37s	36s	36s	37s	37s	35s	35s	35s	33s	31s

Table 8.1 Time taken by a Bidder agent to visit all known/unknown auction sites

[1] The time taken for the Bidder agent to visit all ten auction sites is longer than in the first tests. See Section, Variable Number of Auction Sites in the IAS where it took 37 seconds to visit 10 auction sites and not 29 seconds as measured previously. The reason being that to check that the Bidder agent was working correctly status information was output to the screen.

Discussion

The results in table 8.1 show how long it took the second Bidder agent to visit and discover all ten auction sites. The results are rather surprising; it appears the Bidder agent takes slightly less time when it has an incomplete schedule and can discover new auction sites. One explanation for this is that the Bidder agent interpreter operates faster when it is asking for new auction addresses than when it follows its own schedule. When the Bidder agent follows its own schedule, the interpreter has to read and check the auction references in the Bidder agent script and pick the one that has a TODO status. As the Bidder agent visits more auction sites, the interpreter would need to check through more references to find one that has a TODO status. However, when a Bidder agent asks for a new auction address, the interpreter updates the Bidder agent script with the new auction reference by inserting it before all the other auction references in the schedule. Therefore, the first auction reference it picks is always the new one.

Variable Numbers of Bidder Agents in the IAS

The tests covered in figure 8.3 assume that there are three auction sites in the IAS. All are correct and running and every Bidder agent has a schedule listing all three sites. These tests should show how increasing the number of running Bidder agents in the IAS affects how long Bidder agents take to visit auction sites within the IAS.

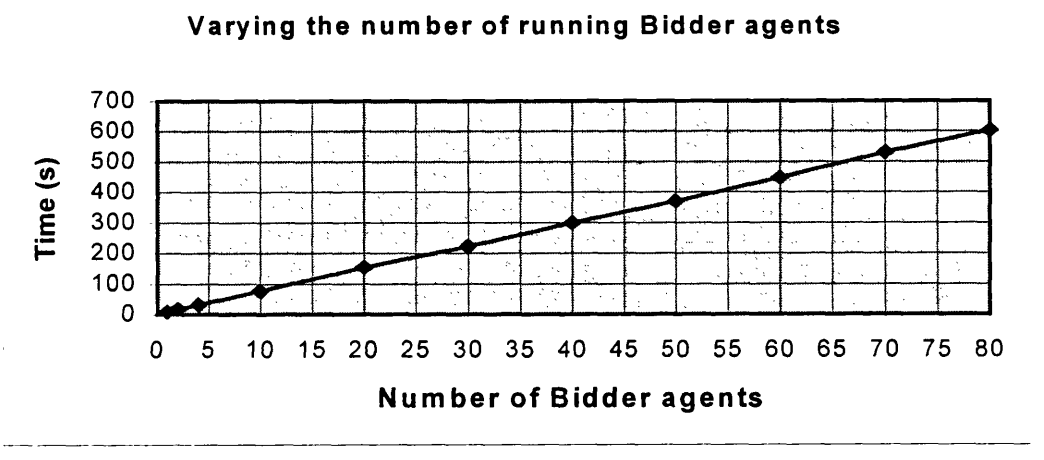


Figure 8.3 Time taken by all Bidder agents to visit all three auction sites

Discussion

The number of Bidder agents that could run concurrently was limited to eighty. Above this number the IAS become unstable. Since, there is no reason why an unlimited number agents cannot run concurrently, it is probably a system limitation, i.e. the operating system would only

allow so many concurrently running threads or it is Java memory limitations. However, this could possibly be corrected by using an operating system that could manage a far greater number of concurrently running threads, e.g. UNIX on Sun workstations.

The results in figure 8.3 show that the more Bidder agents in the IAS the more time it takes the Bidder agents to complete their schedules. This is expected. Although, the more Bidder agents there are, the less time (only very slightly) it takes each Bidder agent to complete its schedule. However, when viewed graphically it can been seen that this decrease is insignificant The graph shows that effectively the time taken by increasing numbers of agents to complete their schedules is linear (for more than 4 Bidder agents). In any case, the times taken for the Bidder agents to complete their schedules with various numbers of concurrently running Bidder agents is fast in comparison with current Web-based auctions which often take hours or days to complete.

8.1.2 Testing for Auction Duration

The following two factors are assessed to see how they affect the time taken to automate auctions:

- Number of Bidder agents;
- Auction method.

The number of Bidder agents should have a direct influence on an auction's duration and so should the auction method. Since in English and Dutch auctions, the auctioneer often asks bidders for their bids many times before the auction ends. Whilst in sealed-bid auctions, bidders only submit their bids once. However, an auction's type should have an insignificant affect on an auction's duration. The reason for this is that the only difference between PV and CV auctions is that Bidder agents are required to make an additional decision (i.e. scale-down). For this reason, only the first two factors are analysed.

Variable Numbers of Bidder Agents in the IAS

The tests covered in figure 8.4 assume that all Bidder agents registered at the auction participate in the bidding. To ensure this, an English-PV auction is used with the starting price = 100, reserve price = 120 and the bidders' maximum bids ranging from 120 up to a maximum of 520. The auctioneer starts bidding at 100 and after each round, increases the asking bid by five until the bidding stops. These tests should show how increasing the number of running Bidder agents in the IAS affects an auction's duration.

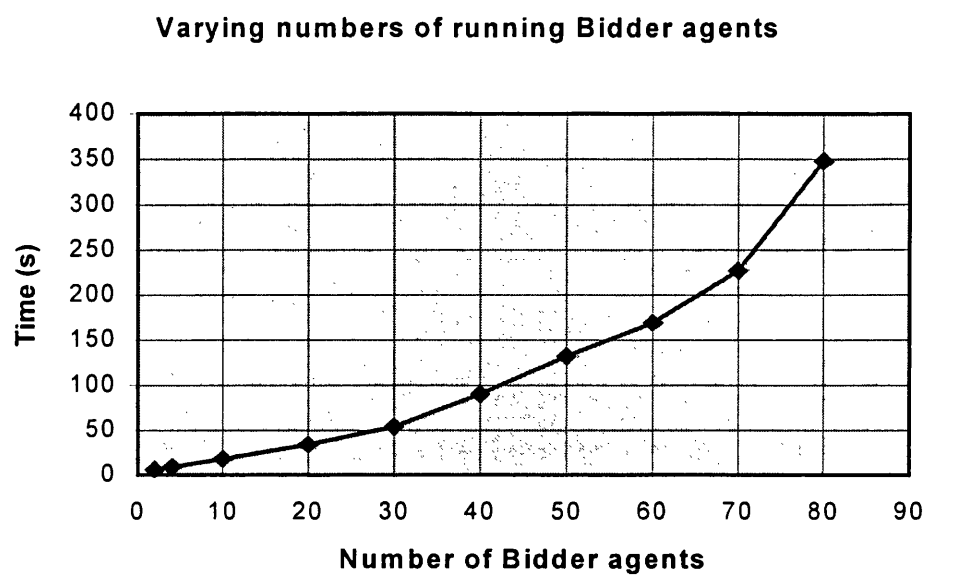


Figure 8.4 Duration of auctions with different numbers of Bidder agents

Discussion

The results in figure 8.4 show how increasing the numbers of Bidder agents within an automated auction increase the time taken to complete the auction process. This is expected for two reasons. As more Bidder agents participate within an auction, the more computer resources are required. In addition, as more Bidder agents participate in the auction, the more auction rounds are required for the auction to finish. The time taken to automate an auction per Bidder agent shows an interesting pattern. Generally, as the number of Bidder agents increases the longer it takes to automate the auction per Bidder agent i.e. the change in the curve's gradient is positive. A possible explanation for this is: as more agents attempt to access the shared-variables in the auctioneer monitor, agents have to keep switching between waiting and running states wasting more and more time as the numbers of agents increase. In effect, the system starts to jam with waiting agents. In any case, the time taken by the agents to automate auctions is measured in minutes. This is an acceptable duration for a real-time auction system and if an auctioneer scheduler system was used to control and prioritise the Bidder agents, the times could reduce.

Auction Methods

The tests covered in table 8.2 assume that there are 30 participating Bidder agents in the auction. In addition, the tests assume that the auction type is PV. For the FPSB and Vickrey auctions, the tests assume the reserve price is 120 and the Bidder agents have a range of bids from 120 to 270. For the English auction, the test assumes the starting price is 100, the reserve price is 120 and

bidders' valuations range from 120 to 270. Finally, for the Dutch auction, the test assumes that the starting price is 320, the reserve price is 200 and the bidders' valuations range between 120 and 270. For the English and Dutch auctions, bid increments and decrements are set to five. These values were chosen so that active bidding could take place.

Method	FPSB	Vickrey	English	Dutch
Time (m/s)	8s	8s	54s	21s

Table 8.2 Duration of auctions using different auction methods

Discussion

The results in table 8.2 show that the English auction takes the longest followed by the Dutch and the Sealed-Bid methods. Typically, this would be the case, since in sealed-bid auctions bidders only submit their bids once whilst in Dutch and English auctions the process of bidding can be much longer. However, the length of auction is particularly dependent on the starting prices, reserve prices, and the bids. In any case, the automated auctions take less than a minute even with thirty Bidder agents. These tests demonstrate the IAS would be a good way to buy and sell goods efficiently and quickly using fully automated Bidder and Seller agents.

8.2 Evaluating the Auction Simulator

This section evaluates the auction simulator and its Fuzzy-Genetic Algorithm (FGA). There are three sub-sections. The first sub-section lists all the parameters that can be set by the user to configure their auction environments and the FGA. In addition, it describes some implicit assumptions. The second sub-section investigates what affect the parameters have on evolving Bidder agent strategies, i.e. convergence and levels of fitness attained. The third sub-section investigates bidding and selling strategies evolved in specific auction environments.

8.2.1 Assumptions

The auction simulator enables users to evolve their agent strategies by setting various parameters and running the FGA. Most of these parameters have been explained, see table 5.1 in Chapter 5. However, explanations of auction rounds and the various populations of fixed and evolving agents are given here. The FGA uses populations of agents, auction rounds, and generations to evolve agent strategies. The FGA consists of four agent populations within an auction, see table

8.3. The four populations are fixed strategy bidders[2], evolving strategy bidders, fixed strategy sellers and evolving strategy sellers. So in any one auction, there will be a certain number (zero included) of fixed and evolving strategy agents.

Agent Populations in one auction	Fixed Bidders	Evolving Bidders	Fixed Sellers	Evolving Sellers
Scenario 1: Evolving Bidder with Fixed Bidder Opponents and Seller	>=0	1	1	0
Scenario 2: Evolving Seller with Fixed Bidders	>=1	0	0	1

Table 8.3 Auction scenarios and their agent populations

In the first auction scenario, the GA population consists of a fixed number of evolving Bidder agents. Each evolving Bidder agent within the population plays (>=2) auctions (a round) with its fixed strategy bidders and fixed strategy seller. The population of evolving Bidder agents is then used to generate the next generation. In the second scenario, the GA population consists of a fixed number of evolving Seller agents. Each evolving Seller agents plays (>=2) auctions (a round) with its fixed strategy bidders. After all the Sellers have played their round of auctions, the population of Seller agents is used to create the next generation. In both scenarios, this continues until all the generations have been evolved.

Besides these configurable parameters, some implicit assumptions are assumed for all auction environments. In particular, the Bidder and Seller agents are risk neutral. Only one item is auctioned at a time. There are no additional royalties or costs incorporated into the auction bids. Bidder and Seller agents attempted to maximise their profits.

8.2.2 Investigating the Fuzzy-Genetic Algorithm Parameters

This section investigates how the FGA parameters affect the average fitness of agents in an evolving population, i.e. the average amount of money they have after playing their round of auctions. The motivation for investigating these parameters is to find a set of parameter values that could be used by users to evolve agent strategies of an acceptable standard in the quickest time. Two important checks used in the assessments are:

2 Fixed strategy bidders and sellers are not really agents.

- How many generations does it take to before an agent population reaches peak fitness?
- What is the maximum average fitness achieved by an agent population in all the generations?

The seven parameters investigated were[3]:

- Number of auctions played by each agent in an evolving population;
- Population size (number of agents in the evolving agent population);
- Number of generations;
- Mutation and Crossover probabilities;
- Gene alleles;
- Fuzzy set partitioning.

The following parameters were kept fixed in all the tests that assessed the above parameters:

- Auction Scenario 1 : only bidding strategies are evolved;
- Auction method is FPSB;
- Auction type is PV;
- Evolving Bidder agent's private value is set at 200;
- Minimum and maximum expected bidder valuations are set to 100 and 200 respectively;
- Minimum and maximum bidder valuation ranges are set to 0 and 50 respectively;
- Number of bidder opponents in the auctions is set to 10;
- Evolving agent's initial money credit = £1000.

Most of these values are not important to the assessment. However, they represent a realistic auction. For example, setting number of bidder opponents in the auction to ten is realistic rather than zero or a ten thousand. Equally, it is realistic to have an auction in which the Bidder agent values the auctioned item at 200 and believes the other bidders expected private values lie between 100 and 200.

Auctions Played Per Round

The tests in figure 8.5 show how the number of auctions played by evolving agents affect an agent population's average fitness. In particular, the *average fitness* attained by the population per 100 auctions played after 100 generations is shown for each test. The other parameters are fixed as follows:

---

3 The parameter, 'number of bidder opponents' is assessed in the section, Evolving Bidding, and Selling Strategies.

- Number of generations = 100;
- Agent population = 100;
- Crossover probability = 0.01;
- Mutation probability = 0.9;
- Gene alleles in Bid Decision Chromosome can take any integer value in {0,1,2,3,4};
- Six fuzzy sets are used to partition 'expected bidder valuation' and five fuzzy sets are used to partition 'bidder valuation range'.

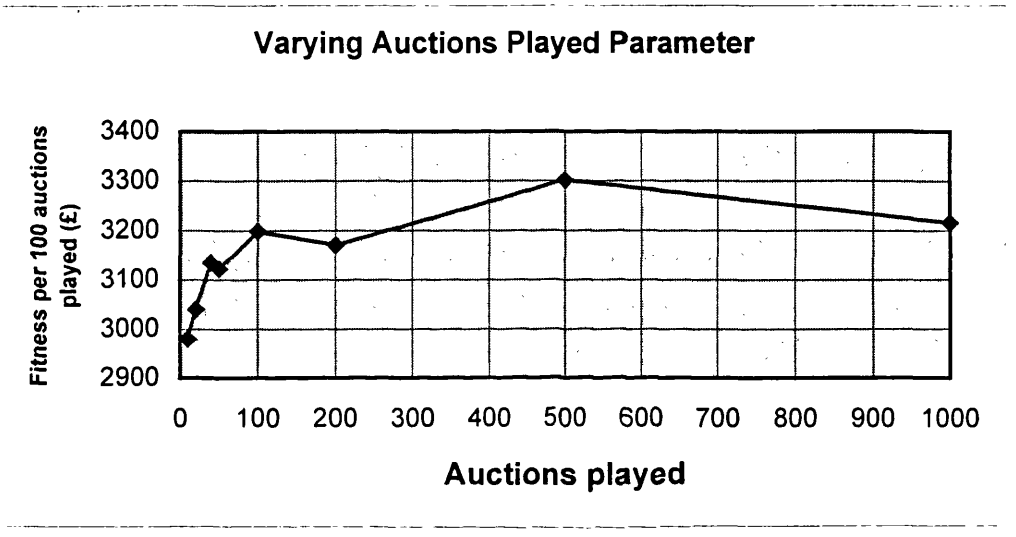


Figure 8.5 Assessing the auctions played parameter

Discussion

The results in figure 8.5 show the average fitness achieved by the agent population per 100 auctions played. The graph gives some indication of how sensitive the FGA is to changing the auctions played parameter. It shows that an agent's fitness increases rapidly when it plays more auctions. However, the increase diminishes when an agent plays more than 100 auctions per session. Therefore, as a compromise between evolving fitter populations and keeping the FGA's running time to a minimum, the auctions played parameter is set to 100. This is used in most of the remaining tests.

Population Sizes and Number of Generations

The tests in figure 8.6 show how the population size and number of generations affect an agent population's average fitness. In particular, the population's average fitness after every ten

generations is shown for various population sizes. The other parameters are fixed at the same values as the previous tests and 'auctions played per round' is set to 100.

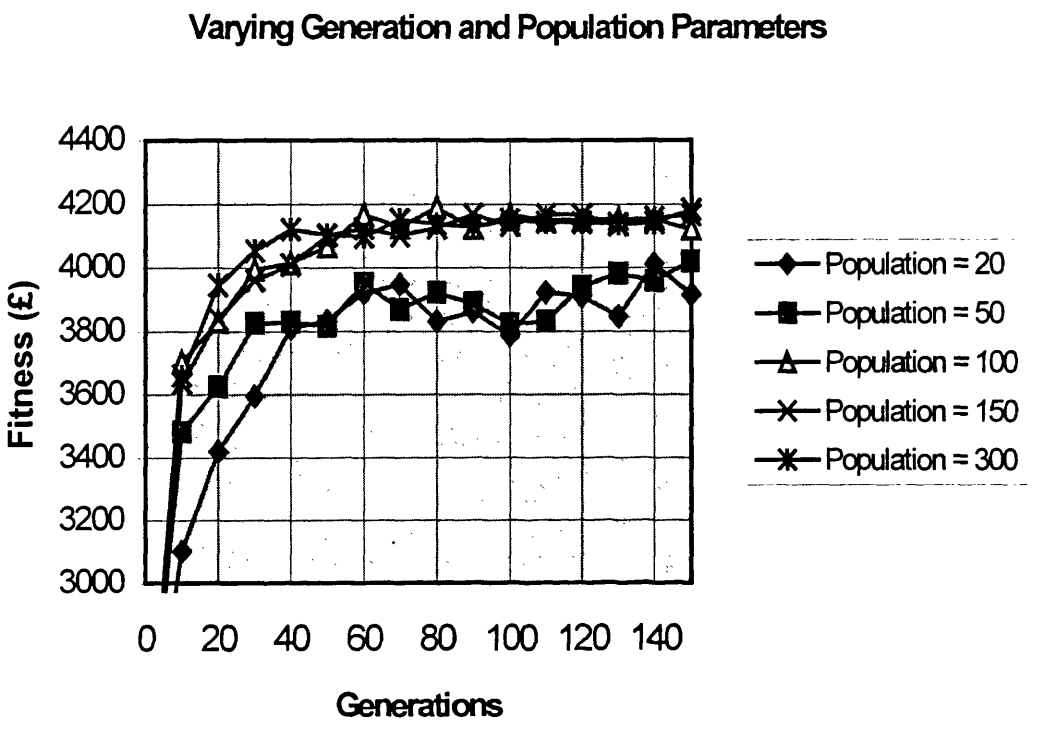


Figure 8.6 Assessing the population size and number of generations parameters

Discussion

The results in figure 8.6 show generally as the population size increases its fitness increases. Therefore, the FGA evolves, on average, fitter agents if more agents are used in the population. However, the average fitness between populations of different sizes tends to converge. For example, after 150 generations, the population with 100 agents is only marginally less fit than populations with greater population sizes. Therefore, a population size of 100 appears to be a reasonable choice in evolving acceptably fit agents in the quickest time. In addition, as the number of generations increase, the agent population's average fitness increases. However, for populations with larger sizes the improvement in fitness reduces after 100 generations. In other words, the last 50 generations appear to have less affect on improving the population's fitness. Therefore, a population size of 100 and an FGA that runs for 100 generations appears to be acceptable. These values are used in the remaining tests.

Mutation and Crossover Probabilities

The tests in table 8.4 show how the crossover probability and mutation probability parameters affect an agent population's average fitness. However, experience shows that GA(s) tend to perform better when crossover probabilities are high (around 0.9) and mutation probabilities are low (around 0.01) [GOL89]. To demonstrate this, the population's average fitness after 100 generations is shown together with the maximum average fitness in all one hundred generations with the generation number in brackets for some sample probabilities. The other parameters are fixed using the same values as the previous tests.

Fitness after every 20 Generations	**Crossover Probability = 0.7**	**Crossover Probability = 0.8**	**Crossover Probability = 0.9**
Mutation Probability = 0.0025	4043 (100)	4072 (100)	4121 (100)
Maximum	4108 (87)	4100 (98)	4122 (94)
Mutation Probability = 0.01	4033 (100)	4101 (100)	4063 (100)
Maximum	4072 (96)	4101 (100)	4131 (93)
Mutation Probability = 0.04	3939 (100)	3961 (100)	3878 (100)
Maximum	3973 (96)	3995 (92)	4006 (71)

Table 8.4 Assessing the crossover probability and mutation probability parameters

Discussion

The results in table 8.4 show that as the crossover probability increases from 0.07 to 0.09, the population's average fitness increases. However, as the mutation probability increases (by a factor of four each time), the maximum average fitness for all populations is (only slightly) higher when the mutation probability is 0.01. However, populations with the highest mutation probability (0.04) do the worst. Therefore, setting the crossover probability to 0.9 and the mutation probability to 0.01 appeared to give the FGA a good chance of evolving fitter agents. These values were used in the remaining tests.

Gene Alleles

The tests in figure 8.7 show how gene alleles affect an agent population's average fitness. In particular, the evolving agent's genes can take any integer value in the interval [0, n] where n>=1. An agent population's average fitness is shown for various allele values. However, in

these tests, genes only take single digit integer values. The other parameters are fixed at the same values as the previous tests except 'auctions played per round' is set to 200.

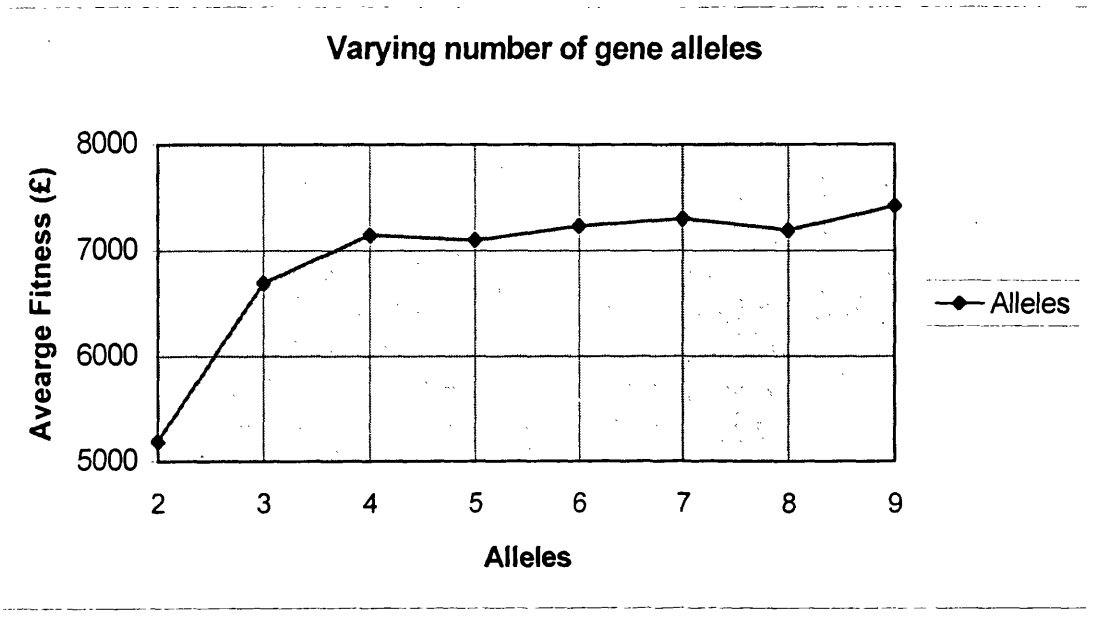


Figure 8.7 Assessing the gene allele parameter

Discussion

The results in figure 8.7 show those agent populations, which have genes that can take a greater variation of allele integer values, are generally fitter. In particular, the average fitness increases dramatically when the genes can take integer values in [0,m] such that m>=3. However, the average fitness does not consistently increase. Even so, the agents that had gene alleles that could take a greater range of integer values did the better because a greater spread of allele values enabled the Bidder agents to bid more accurately. On the other hand, increasing the allowable values for gene alleles increases the number of potential strategies enormously and the FGA may find it more difficult to find good strategies. So, on balance, alleles could take five integer values, i.e. integer values in interval [0,4] (or {0,1,2,3,4}) are used in the remaining tests. This reduces the number of potential strategies that the FGA needs to search but appears to give good fitness results.

Fuzzy Partitions

The tests in table 8.5 show how the fuzzy partition parameters affect an agent population's average fitness. Since the auction type is PV, only 'expected bidder valuation' and 'bidder valuation range' is partitioned. Three different numbers of fuzzy sets are used to partition

'expected bidder valuation' and three more to partition 'bidder valuation range'. The population's average fitness after 100 generations is shown. The other parameters are fixed at the same values as the previous test except 'auctions played per round' is set to 100 and gene alleles can take integer values from [0,4].

Fitness after every 20 Generations	3 Fuzzy Sets Partition Expected Bidder Valuation	6 Fuzzy Sets Partition Expected Bidder Valuation	10 Fuzzy Sets Partition Expected Bidder Valuation
3 Fuzzy Sets Partition Bidder Valuation Range	3702	4150	4217
5 Fuzzy Sets Partition Bidder Valuation Range	3720	4108	4250
10 Fuzzy Sets Partition Bidder Valuation Range	3653	3911	3842

Table 8.5 Assessing the fuzzy partitioning parameters

Discussion

The results in table 8.5 show if 'expected bidder valuation' is partitioned with a greater number of fuzzy sets then the average fitness after one hundred generations increases. This is expected because as 'expected bidder valuation' is partitioned with additional fuzzy sets, an auction's state can be described more precisely. This enables Bidder agents to use strategies that are more specific to each auction state. On the other hand, increasing the number of fuzzy sets used to partition the 'bidder valuation range' appears to have less affect. As more fuzzy sets are used to partition the 'bidder valuation range', Bidder agents perform less well. However, the finer the fuzzy partition the more strategies the FGA has check through. Therefore, making the fuzzy partition finer enables the FGA to check for strategies that have greater precision but there are many more strategies to check. Therefore, these results show that for 'expected bidder valuation' a finer partition more than makes up for the greater number of strategies. However, for 'bidder valuation range' a finer partition is outweighed by the affect of a greater number of strategies. This implies it is more important to know 'expected bidder valuation' more precisely than knowing 'bidder valuation range'. In any case, coarse partitions are used to evolve bidding and selling strategies. This reduces the number of strategies the FGA has to evolve and more importantly, it should make the strategies easier to analyse. This would make assessing whether the FGA evolves good bidding and selling strategies easier too. Therefore, three fuzzy sets are used to partition 'expected bidder valuation', three to partition 'bidder valuation range' and two to partition 'expected bidder scale-down'. Therefore, Bidder agents use nine states to partition the auction state space for a PV auction and eighteen fuzzy sets to partition a CV auction.

8.2.3 Evolving Bidding and Selling Strategies

This section assesses the bidding and selling strategies that are evolved using the FGA. In particular, three sets of strategies are evolved. Firstly, bidding strategies are evolved for the eight auction method/types in which the number of bidders is kept fixed. Secondly, bidding strategies are evolved in auctions in which the number of bidders in the auction varies. Thirdly, selling strategies are evolved for the eight auction method/types in which the number of bidders is kept fixed. The parameters used in the assessments are fixed as follows:

- Number of generations = 100;
- Population of evolving agents = 100;
- Number of auctions played per round = 100;
- Crossover probability = 0.9 and Mutation probability = 0.01;
- Evolving agent's initial money credit = £1000;
- In all decision chromosomes gene alleles can take integer values in interval [0,4];
- Three fuzzy sets are used to partition 'expected bidder valuation', three fuzzy sets partition 'bidder valuation range', and two fuzzy sets partition 'expected bidder scale-down'.

The auctions played per round by each agent, the population size and the number of generations are all set to 100 for two reasons. Firstly, the population's average fitness appears to level off after increasing these values. Secondly, the time taken by the FGA to complete its run of generations takes longer if the values are greater. The crossover and mutation probabilities are set at their values because the FGA appeared to evolve fitter populations. Gene alleles can take five integer values and fuzzy partitions are made coarse because this would reduce the numbers of auction states and thereby make it easier for the FGA to evolve strategies. In addition, the evolved strategies would be easier to analyse. The auction assumptions used in all auctions are:

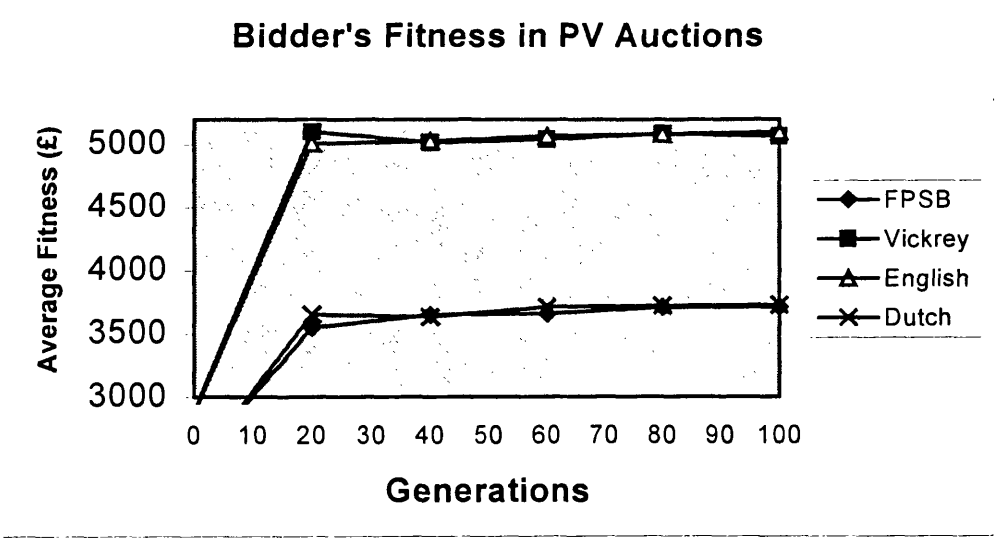
- Minimum expected bidder valuation = 100;
- Maximum expected bidder valuation = 200;
- Minimum bidder valuation range = 0;
- Maximum bidder valuation range = 50;
- Minimum expected bidder scale-down = 10 (CV auctions);
- Maximum expected bidder scale-down = 20 (CV auctions);
- Bidder agent's private value = 200 and Seller agent's private value =100 (PV auctions).

These assumptions are chosen to reflect realistic auctions. For example, in a PV auction, a bidder could privately value the item at 200 because he knows its value precisely. The bidder could also

believe that the other bidders' private values have an expected value between 100 and 200 and the range of values could be between 0 and 50. Therefore, there are four extreme auction situations. The first situation is that the bidders' values have an expected value of 100 and a range of zero; i.e. all bidders value the item at 100. The second situation is that the bidders' values have an expected value of 200 with a range of zero, i.e. all value it at 200. The third situation is that the bidders' values have an expected value of 100 with a range of 50; i.e. bidder values lie uniformly between 50 and 150. The fourth situation is that the bidders' values have an expected value of 200 with a range of 50; i.e. bidder values lie uniformly between 150 and 250. Between these extremes, various other auction situations can arise. The FGA attempts to evolve good strategies for all these auctions.

Bidding Strategies

Bidder agent strategies are evolved for twelve different auction environments. The first four evolve bidding strategies for a PV auction using the four auction methods keeping the number of bidders fixed. The second four auction environments evolve bidding strategies (including scale-down) for CV auctions in the four auction methods. In all eight auctions, the tests assume ten bidder opponents bid in the auctions. The remaining four auction environments assume the auction type is PV and the auction method is FPSB. However, the number of bidder opponents is varied to see if this has any affect on the bidder strategies evolved. In all the auctions, reserve prices and starting prices are unimportant. In figure 8.8, the population's average fitness is shown after every twenty generations for all auction method/types.



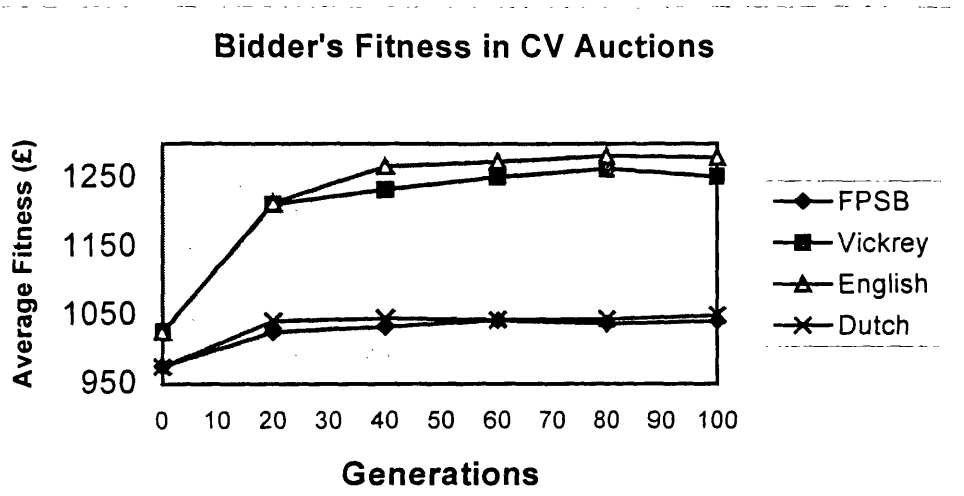


Figure 8.8 Bidder agents' fitness for the auction methods and types

The results in table 8.6 show the bidding strategies of the fittest Bidder agent in the last generation. For the PV auctions, the evolved Bid decision chromosomes are shown and for CV auctions, both the evolved Bid decision and Scale-down decision chromosomes are shown.

Bidding Strategies of fittest Bidder	FPSB	Vickrey	English	Dutch
PV Auctions Bid Chromosome	233-343-322	444-444-434	334-444-443	233-334-322
CV Auctions Bid Chromosome	341212-231412 444312	443300-444423 343433	443411-444413 444412	102002-234204 344403
Scale-down Chromosomes	231442-413201 001113	004021-000013 002004	000144-012012 000030	434112-102120 203031

Table 8.6 Bidding strategies for auction methods and types

Discussion for PV Auctions

The results in figure 8.8 show that in PV and CV auctions, the Bidder agent population's average fitness is similar for FPSB and Dutch auctions and similar for Vickrey and English auctions. This is consistent with the results obtained in [RAS96]. Since, no information is gained by Bidder agents in both FPSB and Dutch auctions they are strategically the same. Similarly, Vickrey and English auctions are strategically the same because the bid price paid by the winner is the second

highest bid[4]. Interestingly, in the CV auctions, the results indicate the same pattern. However, the population's average fitness in CV auctions is much less than in PV auctions. The reason being in PV auctions, bidders know the value of the auctioned item and can ensure that they never lose money. On the other hand in CV auctions, bidders do not know the true value of the auctioned item. Bidders have to estimate its value using their valuations and these may not be good estimates. The bidder that over values the most wins the auction. Therefore, the winning bidder is more likely to overbid and make a loss.

As for the number of PV auctions won in a round of one hundred auctions played, roughly the same number of auctions were won by Bidder agents in the FPSB and Dutch auctions. Similarly, for the Vickrey and English auctions. However, far more auctions were won (over 90%) in the English and the Vickrey auctions compared with fewer (roughly 70%) were won in the FPSB and Dutch auctions. This makes sense since the Bidder agent strategies suggest that Bidder agents should bid less in FPSB-PV and Dutch-PV auctions than in Vickrey-PV and English-PV auctions. Therefore, Bidder agents are likely to win more auctions in Vickrey-PV and English-PV. In addition, Bidder agents make more profit in these auctions.

The results in table 8.6 show the evolved chromosomes for the fittest Bidder agent. These encode what the Bidder agent should bid given the auction state, i.e. 'expected bidder valuation' and 'bidder valuation range'. The Bid decision chromosome for the FPSB auction is 233-343-322. The first three genes represent the bidding strategies when the auction state is 'low expected bidder valuation', and 'low, medium and high bidder valuation ranges' respectively. The second set of three genes represents the strategies for auction states 'medium expected bidder valuation' and 'low, medium and high bidder valuation ranges'. Finally, the last three are strategies for the auction states 'high expected bidder valuation' and 'low, medium and high bidder valuation ranges'. The gene values indicate the level of bidding. This is derived from the bidder valuations (see sections 6.2.2 and 7.2.2 for details). Higher gene values imply higher bids. So that a gene value of zero indicates bid very low, 1 indicates bid low, 2 indicates bid medium, 3 indicates bid high and 4 indicates bid very high relative to the bidder valuations. Therefore, the strategies for the FPSB-PV auction are to bid medium to high for most of the auction states. Similar strategies were evolved for the Dutch-PV auction, i.e. 233-334-322. This is consistent with the theoretical results described in [RAS96], [VIC61] since both auctions are strategically equivalent. However, the evolved chromosomes for the Vickrey and English auctions are 444-444-434 and 334-444-443. These strategies suggest that the Bidder agent on average should bid 'very high'. Since, the evolving Bidder agents must choose a bid less than maximum bidder valuation (in these tests 250) and their private value is 200 then gene values of 4 imply 'bid as close to private value'.

---

4 In English auction, a bidder need only bid the second highest bid plus a small amount to win.

Bidding close to your private value in Vickrey-PV and English-PV auctions is optimal strategy as described in [RAS96], [VIC61][5]. In general, it appears that the Bidder agents should bid less in FPSB-CV and Dutch-PV auctions than in Vickrey-PV and English-PV auctions. This is also consistent with theoretical results in [RAS96]. Although, the differences in the bids should not be so great, i.e. the bids for FPSB-PV and Dutch-PV should be marginally less. However, making gene alleles take higher integer values may show this, though the number of possible strategies increases exponentially.

Discussion for CV Auctions

The CV auctions are more complex to analyse because a third factor, 'expected bidder scale-down' was introduced. This time there are two decision chromosomes, one for the bid and one for the scale-down to reduce the bid. Both chromosomes consist of 18 genes. The first six genes represent strategies for auctions with the state 'low expected bidder valuation'. The second six genes represent strategies for the auction with states 'medium expected bidder valuation' etc. Out of the first six genes, the first two refer to the auctions with states 'low bidder valuation range'. The second pair of genes represents the auction states 'medium bidder valuation range' and the last two 'high bidder valuation range'. Similarly out of the first two genes, the first one refers to the auction state 'low expected bidder scale-down' and the second gene refers to 'high expected bidder scale-down', and so on for all the genes in the chromosomes. Therefore, each gene represents the strategy for a unique auction state.

As for the number of auctions won in a round of one hundred auctions played, in the FPSB-CV and Dutch-CV auctions, roughly the same number of auctions were won by the Bidder agent. Similarly, for the Vickrey-CV and English-CV auctions. However, far more auctions were won (over half) in the English and Vickrey auctions compared with far fewer (roughly 10-15%) were won in the FPSB and Dutch auctions.

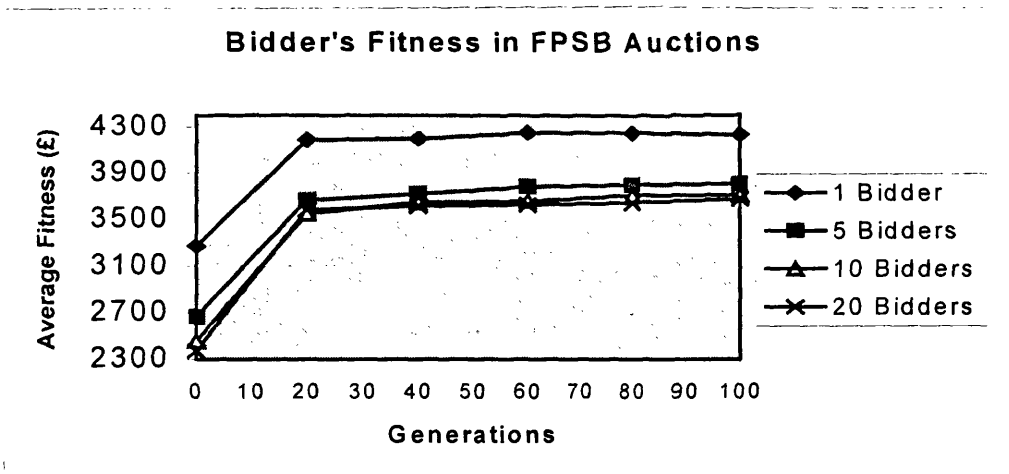
Though the Bidder agents made a profit in all the CV auctions (they did not succumb to winner's curse), the strategies for the CV auctions appear rather complex. Looking at gene alleles in the Bid chromosomes for the FPSB and Dutch auctions, they appear to be lower than for the English and Vickrey auctions. In addition, the alleles of genes in the scale-down chromosomes for the FPSB and Dutch auctions are slightly higher suggesting that the overall bid, i.e. bid minus scale-down should be less than for English and Vickrey auctions. Therefore, the bidding and scale-down strategies evolved by this FGA are to bid less and scale-down more in FPSB and Dutch

5 In Vickrey auctions bidding less than your private value would reduce your chances of winning but you still pay the same if you win. However, if you bid more than your private value then you could win the auction but lose money. Therefore, the optimal strategy is to bid your private value.

auctions, compared with Vickrey and English auctions. Analysing the bidding strategies for the Vickrey and English auctions, suggest that 'high' bids should be made for all auction states except when the 'bidder valuation range' is 'high', i.e. genes pairs 5 & 6, 11 & 12 and 17 & 18. Conversely, the scale-down strategies for English and Vickrey auctions suggest scale-down more for auctions with 'high expected bidder ranges'. Analysing the bid and scale-down chromosomes for the FPSB and Dutch auctions reveals no discernible pattern. In any case, [RAS96] theoretical results suggest that the strategies for FPSB-CV and Dutch-CV auctions should be the same. Therefore, the results obtained here are not as expected. A possible reason for this is that in CV auctions the strategy spaces are much larger than in PV auctions (CV auctions have approx. 4 thousand billion strategies and PV auctions have approx. 2 million). Comparing CV to PV auctions, Bidder agents make less profit and win fewer auctions. However, strategies for CV auctions are more difficult to evaluate than for PV auctions.

Varying Numbers of Bidder Opponents

In figure 8.9, the results show the Bidder agent population's average fitness after every twenty generations with various numbers of bidder opponents (to the evolving Bidder agent). In addition, the bidding strategies evolved by the FGA are shown in the form of decision chromosomes.



Bid Chromosomes evolved by the FGA	1 Bidder Opponent	5 Bidder Opponents	10 Bidder Opponents	20 Bidder Opponents
	223-332-222	324-343-333	433-343-343	333-434-344

Figure 8.9 Bidding strategies for auctions with various numbers of bidder opponents

Discussion

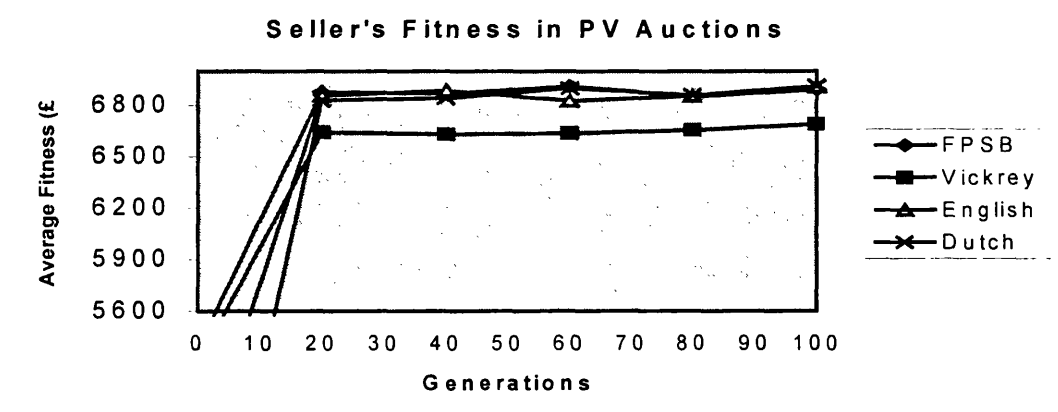
The results in figure 8.9 show that varying the number of bidder opponents in the auction affects the population's average fitness. The more bidders present within the simulated auctions, the

lower the population's average fitness is. This is an expected result, since the more bidders participating, the greater chance one would have a very high valuation and the only way for the Bidder agent to win the auction is to bid very high. The converse is also true. If there is only one bidder opponent present at the auction, there is less chance that the bidder opponent will have a very high valuation. This enables the Bidder agent to bid less and still win the auction. The chromosomes encoding the strategies reflect this. The chromosome for the FPSB-PV auction with one bidder opponent is 2-2-3-3-2-2-2-2-2 whilst for the auction with five or more bidder opponents have chromosomes that contain genes that generally have higher values of three and four. Therefore, these strategies suggest that the greater the number of bidder opponents in an auction, the higher the Bidder agent should bid, i.e. close to their private values. These results are confirmed in [RAS96] (and proved originally by Vickrey[6] [VIC61]).

Selling Strategies

Seller agent strategies (evolving reserve prices and starting prices) are evolved for eight different auction environments. The eight environments are made up of the eight auction method/types. Most of the auction assumptions are the same as for previous tests. The only difference being the value set for the Seller agent's private value. This is set to 100 since a seller is more likely to want to auction their items in an auction that they believe has bidders with higher private values or valuations. General information on selling strategies for the auction methods and types discussed in this section can be found in [AGO96] and [RAS96].

In figure 8.10, the Seller agent population's average fitness is shown after every twenty generations for the various auction methods and types. Finally, in table 8.7, the evolved selling strategies are shown for the auction methods and types.



6 Vickrey formulated the optimal bidding strategy for FPSB-PV auctions as bid = (N*v)/(N+1), (assuming, the bidder opponents valuations are uniform) where N is the number of bidder opponents and v is the bidder's private value. So for N=2, the bidder's optimal bid is half his private value and as N becomes larger, the bidder's optimal strategy is to bid closer to his private value. In other words, the bidder should bid what he thinks the item is worth.

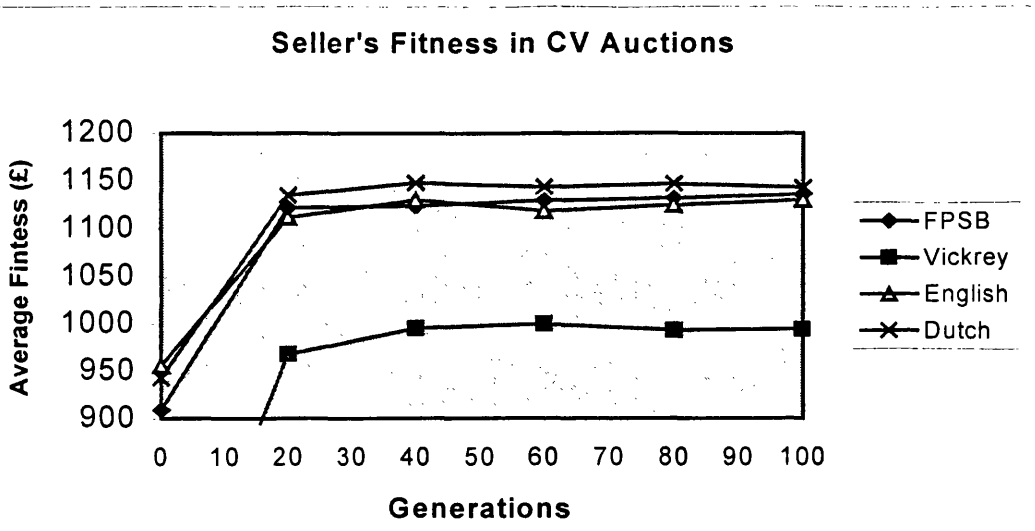


Figure 8.10 Seller agents' fitness for auction methods and types

Selling Strategies of fittest Seller	FPSB	Vickrey	English	Dutch
PV Auctions Reserve Price	100-001-201	000-001-230	001-010-000	100-001-011
Starting Price			101-012-221	343-133-342
CV Auctions Reserve Price	343303-443402 444422	433404-444414 444424	444400-221411 344303	442203-432411 441313
Starting Price			443101-444402 434414	323143-023224 211243

Table 8.7 Selling strategies for auction methods and types

Discussion for PV and CV auctions

The results in figure 8.10 show that in PV auctions the profits made are very similar for the FPSB, English, and Dutch but marginally less for the Vickrey. This is consistent with the revenue-equivalence theorem proved by [VIC61]. This states that in all the four auction methods, the expected revenue should be the same. The results in figure 8.10 for PV auctions show the Reserve Price decision chromosomes predominantly have genes with low values, i.e. 0 or 1. This strategy implies setting reserve prices low or very low. However, the starting price chromosomes are different for the English and Dutch auction methods. In English-PV auctions, the Seller agent's strategy is to set a low starting price but in Dutch auctions, it is to set a higher starting

price. This makes sense; since in English auctions a high starting price could prevent bidding and reduce revenue, or the chance of a sale. Similarly, in Dutch auctions a low starting price may miss the highest bids and reduce revenue.

The results for the CV auctions are interesting because the profits are much lower than in PV auctions. In Vickrey auctions, the Seller agent fails to make any profit. In CV auctions, the revenue-equivalence theorem does not apply. According to [AGO96], the revenue generated should be highest for English auctions followed by Vickrey then FPSB and Dutch auctions equally. However, this assumes bidders observe each other in the English auction and this is not true of this simulation. Never-the-less, the results show FPSB, English and Dutch auctions yielding the same revenue and Vickrey the least. Therefore, in this respect, the results are not consistent with auction theory. Another interesting observation is that all the auctions generate similar profits except for the Vickrey auction.

The results in table 8.7 for CV auction (starting and reserve price) strategies are complex and difficult to analyse. The strategies generally imply that reserve prices are set high or very high for many of the auction states. The one clear exception is for the fifth gene on the Reserve Price decision chromosome for all the four auction methods. This is when the auction has bidder valuations described as 'low expected bidder valuation', 'high bidder valuation range', and 'low expected bidder scale-down'. The evolved strategies suggest that the reserve price should be set very low. In contrast, to the starting price strategies in PV auctions, the starting price strategies for CV auctions are the converse. In English auctions, the starting price is set high to very high for many of the auction states but in Dutch it is set medium to high. These unexpected results are likely to be explained by the fact that in CV auctions the strategy spaces are much greater than for PV auctions (CV auctions have approx. 4 thousand billion strategies and PV auctions have approx. 2 million). Therefore, the FGA finds it difficult to find good/optimal strategies in CV auctions.

The results obtained are mixed. Some are confirmed by auction theory, others contradict it. Therefore, the results demonstrate that the FGA has been of limited success. When strategy spaces are small (in PV auctions), the results are good but in larger strategy spaces (in CV auctions), the results are not so good. However, most bidding and selling strategies made some profit given that the auctions were simplified models of real auctions. This does not imply that they would do well in real agent auctions where the assumptions could be relaxed but it does imply that the FGA from an initial random selection of strategies evolved better strategies. Therefore, these results give researcher(s) a basis to explore the FGA further.

Chapter 9 - Conclusion

This chapter concludes the thesis in three sections. The first section summarises the thesis. It describes the motivation for designing Adaptable Mobile Agents (AMA). In addition, it describes how AMA(s) are used to automate auctions by constructing an Internet Auction System (IAS). The second section describes the tests and results used to evaluate the IAS and its auction simulator. The third section outlines some ideas for further work.

9. 1 Thesis Overview

This thesis aimed to automate various auction methods over the Internet by creating a novel dynamic and distributed auction system. Primarily, this system should enable users (as sellers) to sell their items from their computers by choosing the auction method and deciding when the auction should start. In addition, the system should enable users (as bidders) to visit and bid in auctions held on sellers' computers.

The methods used to build and implement the IAS were software agents. In particular, this thesis was interested in designing novel tailor made software agents that were mobile, manageable, autonomous, and adaptable. These were called Adaptable Mobile Agents (AMA). The main reason for exploiting these agents was to see how suitable they would be at automating Internet auctions.

For example, users (as sellers) could automate the selling part of an auction on their computers using immobile Seller agents. In addition, they could adapt their agents' selling strategies, configure and run them The Seller agents would then fully automate their auctions and simplify the selling process over the Internet. As Seller agents automate auctions, their users could monitor them in real-time. Moreover, users (as bidders) could automate the searching for and bidding in remote auctions (held by Seller agents) using mobile Bidder agents. Similarly, they could adapt their Bidder agents' bidding strategies, configure, and run them by sending them to visit and explore remote auctions. The Bidder agents would then register with the most favourable auction and automate the bidding when the auction started. Finally, if the Bidder agent were successful (winning the auction) it would return home to its user, otherwise, it would continue to search for other auctions.

Agent Mobility and Management

The motivation for creating manageable mobile agents was to use them in the construction of distributed systems like Internet auctions. Agent mobility was implemented as a network of agent client/servers and web servers connected by temporary socket connections and HTTP post requests. This infrastructure enabled agents to move from one computer to another within the agent network. The web server was used to route agents enabling mobile users to join and leave an agent network from any computer within the network. Users used GUI(s) to create, run, and monitor their agents. The agents themselves informed their users of where they were and what they were doing.

Agent Automation and Adaptation

The motivation for creating autonomous agents was to use them to automate processes. In the auction system, the selling and bidding processes were automated using Seller and Bidder agents. These agents were modelled as finite automata and written in simple SGML-based scripting languages to standardise the agents. These marked up all the agent's information and states. Agent interpreters were used to extract information from the scripts and run them as finite automata. However, the agents never communicated directly. Instead, they used auctioneer monitors to mediate inter-agent communications. For example, Seller agents used their auctioneer monitors to accept Bidder agents registering to attend the auction. In addition, when the Seller agent's auction had started, it used its auctioneer monitor to set asking bids, read the Bidder agents' answers and set the auction outcome. This synchronised both Bidder and Seller agents enabling them to automate local auctions very quickly. Formalising the multithreaded auction process using an auctioneer monitor would increase user confidence that auctions would be fair, efficient, and stable.

The motivation for designing adaptable agents was to enable users to evolve their agents in a simulator. If the simulator could model the agent's environment as a game in which the agents were the players then the agent strategies could be tested and improved using techniques like Fuzzy Logic and Genetic Algorithms. Fuzzy sets could be used to partition the agents' information spaces, and chromosomes and genes could used to encode the agent strategies. These strategies could then be evolved in simulated games using robust Fuzzy-Genetic Algorithms (FGA). The evolved strategies would provide users with agent strategies that had been specifically adapted for their chosen environment. Users would then have more confidence in allowing their agents to make decisions on their behalf in real agent systems like Internet auctions.

For example, an auction simulator was created that modelled the auction methods and types as strategic games. Users could use parameters to configure their desired auction environments and evolve bidding or selling strategies. In particular, two auction scenarios were investigated. The first auction scenario assumed that at every auction only one Bidder agent had evolving strategies. The bidder opponents and the seller had fixed strategies. The second auction scenario assumed that the Seller agent had evolving strategies and all the bidders had fixed strategies.

During real or simulated auctions, Bidder agents made decisions on what to bid and scale-down. Seller agents also made decisions on what to set the starting price and reserve prices to. The information (state of the auction) they used to make these decisions was partitioned by fuzzy sets. Their strategies were encoded using chromosomes. Each gene represented a strategy for a particular state of the auction. Since there were potentially a large number of auction states, the fuzzy partitioning reduced the number of strategies the agents required. In addition, it increased the chance that the FGA would evolve good strategies. Users could view their agent population's progress and performance and see how they were improving from one generation to the next. During the last generation, users could choose strategies for their agents to use in real auctions held in the IAS.

9.2 Evaluation

The IAS was evaluated by testing to see if the agents automated auctions effectively. In particular, the IAS was evaluated by testing how long Bidder agents took to navigate around various agent networks and how long the automated auctions took. The auction simulator and its FGA were evaluated in two ways. Firstly, by assessing what effect the parameters had on an evolving population's average fitness. Secondly, by using suitably selected parameters to evolve bidding and selling strategies in various auction environments and comparing them with results from auction theory.

Bidder Agent Navigation

Tests were carried out by constructing the IAS with various numbers of auction sites and Bidder agents on one computer system. Four factors were used to test how long it took Bidder agents to navigate around an IAS. Firstly, the number of auction sites the Bidder agent was required to visit was varied. Secondly, the number of auction sites that were incorrect in the IAS was varied. Thirdly, the number of auction sites that were unknown to the Bidder agent within the IAS was varied. Fourthly, the number of concurrently running Bidder agents in the IAS was varied. The main findings were:

- The more auction sites that a Bidder agent had to visit the longer it took. However, the increase in time taken by the Bidder agent to visit increasing numbers of auction sites was approximately linear.

- Bidder agents successfully navigated around IAS(s) with various numbers of incorrect (wrong addresses or auction sites that were not running) and unknown auction sites (not in its schedule). In either case, the time taken for the agent to complete its schedule by avoiding incorrect auction sites or exploring new ones was similar.

- The times taken for increasing numbers of concurrently running Bidder agents to complete their schedules increased linearly. Therefore, increasing numbers of concurrently running agents did not have an adverse effect on the overall systems performance.

Auction Duration

Similarly, various tests were carried out to assess how the auction method and the number of Bidder agents bidding in the auctions affected an auction's duration. The main findings were:

- The time taken to automate auctions ranged from 6 seconds for auctions with two Bidder agents to 6 minutes for auctions with 80 Bidder agents. The time taken by increasing numbers of Bidder agents to compete in the auctions increased substantially as the number of Bidder agents reached 80. The main reason for this is that more and more agents would be waiting to access the shared variables in the auctioneer monitor and this may not be very efficient for larger numbers of Bidder agents. However, these automated auctions were an improvement on typical Web auctions that took hours or days. Therefore, users could use Bidder agents to search, visit, and bid in many auctions in real-time.

- In the tests carried out, the times taken by the agents to automate, the auctions were measured in seconds and/or minutes. In this respect, the IAS could be used to automate real auctions in real-time.

Evaluating the Auction Simulator

The auction simulator used a FGA to evolve bidding and selling strategies for a variety of auction environments. Both the auction environment and FGA were configured using parameters. The auction simulator and FGA were evaluated in two ways. Firstly, the parameters were investigated to see how they affected an evolving agent population's average fitness. Secondly,

suitably selected parameters were then used to evolve bidding and selling strategies in various auction environments.

Assessing the FGA Parameters

Tests were carried out on the following parameters: number of auctions played per agent, agent population size and number of generations, crossover and mutation probabilities, gene alleles on the agents' decision chromosomes and number of fuzzy sets used to partition an agent's information space. The main findings were:

- It appears that the more auctions played the fitter the evolved agents became. However, the improvement in fitness was marginal after 100 auctions were played. Therefore, each evolving agent played 100 auctions per round for most of the remaining assessments.

- As the size of a population was increased, its average fitness increased. However, for populations of 100 or more, the increase in fitness was marginal. Similarly, the more generations used to evolve agents the fitter the population became. After about 100 generations, the population's average fitness peaked before levelling off. The tests showed that for this FGA, the auction parameters, population size and number of generations should be set to 100 to evolve the fittest agents in the least time.

- In nature, crossover probabilities tend to high and mutation probabilities tend to be low. The greater the crossover and mutation probabilities, the greater the chance that new strategies could be discovered but too high a mutation probability and GA(s) tend to generate random strategies. The tests confirmed this showing that higher crossover probabilities (0.9) and lower mutation probabilities (0.01) enabled the FGA to evolve fitter agents than a FGA that used a lower crossover probability or high or very low mutation probabilities.

- Generally, agents that used more allele values in their chromosomes evolved into fitter agents. However, increasing allele values above four had very little affect on fitness. This was a surprising result since the higher the value the more accurate the agent could bid. However, the greater variation of allele values the greater the size of the strategy making the FGA task of evolving good strategies more difficult. On balance, alleles that could take values from {0,1,2,3,4} for the agent's decision chromosomes were chosen. This would enable the FGA to evolve strategies that would perform well and would be easier to analyse.

- Increasing the number of fuzzy sets used to partition 'expected bidder valuation' enabled Bidder agents to make bids that were more accurate and this improved their fitness.

However, increasing the number of fuzzy sets used to partition 'bidder valuation range' did not improve the population's average fitness. This suggested that making the partition for 'bidder valuation range' finer was outweigh by the effect of increasing the number of strategies that the FGA had to search through. Whilst making the partition for 'expected bidder valuation' finer did improve fitness, implying, that knowing 'expected bidder valuation' more precisely was more important than knowing 'bidder valuation range'.

Evolving Bidding Strategies

Bidding strategies were evolved for all auction methods and types. Each evolving Bidder agent from a population took turns to play 100 auctions. For each auction, only the evolving Bidder agent had evolving strategies all the bidder opponents and the seller had simple fixed strategies. The main results were:

- In PV auctions, the fitness (profit) achieved in FPSB and Dutch auctions *and* in Vickrey and English auctions was approximately the same. In addition, the agents won more auctions in English and Vickrey auctions than in FPSB and Dutch auctions. The strategies for PV auctions were similar for FPSB and Dutch auctions implying that Bidder agents bid high. Similarly, the strategies for English and Vickrey auctions were similar, implying that Bidder agents bid close to their private values but less than in FPSB and Dutch auctions. This was consistent with results from auction theory [RAS86], [VIC61] (since FPSB and Dutch auctions are strategically equivalent and so are Vickrey and English auctions). Overall, the results were encouraging for PV auctions.

- Analysing the strategies for CV auctions was difficult because few discernible patterns were evident in the decision chromosomes. A reason for this was the large increase in the strategy space for CV auctions compared with PV auctions. However, the bid and scale-down strategies suggested that Bidder agents should bid less and scale-down more in FPSB and Dutch auctions than in Vickrey and English auctions. These strategies enabled the Bidder agent to make profits in all the auction methods. In FPSB and Dutch auctions, Bidder agents won about 10-15% of the auctions they played. Whilst in Vickrey and English auctions they were more successful, winning on average about half the auctions they played.

Varying Number of Bidder Opponents

Bidding strategies were evolved in auction environments in which the number of bidder opponents in the auctions varied. The chosen auction method and type was FPSB-PV and the numbers of bidder opponents were 1, 5, 10, and 20 respectively. The main results were:

- As the number of bidders participating increased, the population's average fitness decreased. This was expected because as more bidders were present the greater chance a bidder with a high private value would be bidding.

- The strategies suggested that the more bidders bidding in the auction, the higher the Bidder agent should bid. It should bid closer to its private value to increase its chances of winning without making a loss. These results were consistent with results proved in auction theory [RAS96].

Evolving Selling Strategies

Selling strategies were evolved for all auction methods and types. Each Seller agent in a population took turns to play 100 auctions. In the auctions, only the Seller agent evolved all the bidder opponents had simple fixed strategies. The main results were:

- In PV auctions, the item was successfully auctioned in nearly all the auctions for all four methods. In addition, the revenue made was similar. This matches the results obtained in auction theory, i.e. the revenue-equivalence theorem [VIC61]. The reserve price strategies generally suggested setting the reserve price low or very low for all auction methods and for most auction states. The starting price strategies for the English auction implied setting low starting prices low and for the Dutch auctions setting high starting prices. These strategies made sense because if the Seller agent set starting prices too high in English auctions or set reserve prices too high in any auction then there could be a 'no-sale'. Although, if the Seller agent set lower reserve prices it would still make the Seller agent the same profit.

- The selling strategies evolved in CV auctions were complex. The profits made by the Seller agent were much lower than in PV auctions. In addition, the number of auctions in which the item was sold was far less than for PV auctions. In the FPSB, English and Dutch auctions, the item was sold in about 40% of the auctions but in Vickrey auctions the item was sold in about 20% of the auctions. In CV auctions, English and Vickrey auctions should generate more revenue than the FPSB and Dutch (which should generate the same revenue). The profits or revenues generated in these tests were similar FPSB, English and Dutch auctions, but lower for the Vickrey auction. Though the revenues for the FPSB and Dutch auctions should be the same, they should not be higher than for the Vickrey auctions. Concerning the strategies, they suggested that reserve prices should be set high to very high compared with PV auctions. In addition, the strategies suggested that the starting price should be set to high

for English auctions but low for Dutch auctions. Therefore, some results appeared to be inconsistent with auction theory.

Overall, the results showed that both bidding and selling strategies were successful in that they made profit for their agents (given the assumptions). However, because the strategy spaces for PV auctions were much smaller in size (by a factor of 2 million) than for CV auctions, the FGA was more successful at evolving good strategies in PV auctions than in CV auctions.

9.3 Further Work

The thesis ends with some ideas for further work. In particular, ideas to formalise and enhance the AMA, ideas to improve the IAS, and ideas for new applications.

Adaptable Mobile Agents

Designing generic agents systems is difficult and full of pitfalls [WOO98]. Often the agents are designed in an ad-hoc fashion and forced on unsuitable applications; this applies equally to the AMA. However, by demonstrating that the AMA can be used to automate distributed auctions over the Internet, it is feasible they could be used to automate other electronic markets and electronic commerce systems, see section New Applications. However, for every application, a different SGML-based language (DTD) would need to be specified. If the applications are relatively simple creating simple scripting languages is easy but writing the agent interpreters and monitors may not be. In addition, using Fuzzy Logic and GA(s) to encode agent strategies and designing the simulator could be difficult. Attempting to design tools that automatically generate scripting languages, interpreters, monitors, and simulators appears very difficult but an interesting problem for future researchers on agent-based systems to explore.

Internet Auction System

For the most part the IAS is complete. However, the agent interfaces are rudimentary GUI(s). Perhaps a better way to interface with the auctions is by using animated agents. The problem with animation is transmitting data from the auction to all interested parties. If there is a delay in the information being sent, or it is received out of sequence by the users then their view of the remote auction would be distorted. One solution is to wait for the auction to end and send a log of auction events to each user. Users then replay the auction on their local machines at their desired speed. However, the automated auctions must be fast, lasting seconds rather than minutes. In addition, agent security within the IAS is rudimentary. Any user may read agent

scripts because they are written in ASCII plain text. Therefore, agent authentication or encryption is required to encode all agent scripts before they are sent over the Internet. In addition, the IAS is a multithreaded distributed system that is difficult to check and verify. Undoubtedly, the system requires further analysis for unusual user, system, and network events. In particular, the IAS requires recovery databases to store agents and their states in case of system crashes.

The simulator modelled auctions in a simplistic fashion by making assumptions about how the bidders value the item, i.e. risk-neutral PV and CV auctions. Even so, the evolved strategies were often difficult to analyse. Modelling real auctions by taking into account real bidder behaviour and auctioneer behaviour is far more complex. For example, in realistic auctions the item's value is often unknown and bidders may use various pieces of private and public information to secure winning the auction at a profitable bid. In addition, the opponents may not apply simple fixed strategies. They may evolve their strategies or rely on psychological rather than logical reasoning [CAS67]. Therefore, the next generation of Bidder and Seller agents will need to be more sophisticated, cleverer and behave more like professional bidders and sellers.

New Application Areas

Though the software agents designed in this thesis have been applied to automating Internet auctions, they could be used for other e-commerce systems. The potential for using mobile agents with some embedded intelligence in e-commerce systems is most notable in virtual markets. For example, AMA(s) could be used to buy and sell in virtual stock markets (double auctions). Users as home based small investors could create their mobile agents to participate in virtual stock markets and trade by purchasing or selling shares on their behalf. Similarly, users could let their mobile agents 'go shopping' at virtual stores or conversely let their agents seek potential buyers. Another potential application for mobile agents could be modelling multi-party negotiations between chains of service providers in which each part of the service chain is trying to optimise their revenues. Mobile agents (representing different parts of the service chain) could first locate each other and form a multi-agent service chain. Once all the agent parties are located together, they could negotiate with each other attempting to reach a mutual agreement. Finally, mobile agents do not have to compete in their virtual environments; they could co-operate to perform tasks. For example, many mobile agents could search virtual markets and return with data (market prices, quantities of goods, type of seller or virtual market). This information could then be stored in a database and used by other agents to build up an accurate picture of a particular virtual market. These agents could then visit the virtual market(s) and negotiate with the seller(s) (agent or human) using their personal information (derived from the database) and strategies.

Appendix A - Fuzzy Set Partitioning

This appendix demonstrates how fuzzy sets are used to partition an agent's information space or auction state space. Figure A.1 shows five fuzzy sets partitioning 'expected bidder valuation'. However, all the methods described in this appendix apply equally to:

- Expected Bidder Valuation;
- Bidder Valuation Range;
- Expected Bidder Scale-down.

Let A be the minimum 'expected bidder valuation' and B be the maximum 'expected bidder valuation' for all the bidders in a particular auction. Let p fuzzy sets partition 'expected bidder valuation' then in figure A.1, d = (B-A)/(p-1) = (B-A)/4

and if neighbouring (overlapping) fuzzy sets Q and R have membership functions

$\mu 1 and \mu 2$

then the fuzzy sets over lap s.t.

$$\mu 1(v) + \mu 2(v) = 1 \forall v \in Q \cap R$$

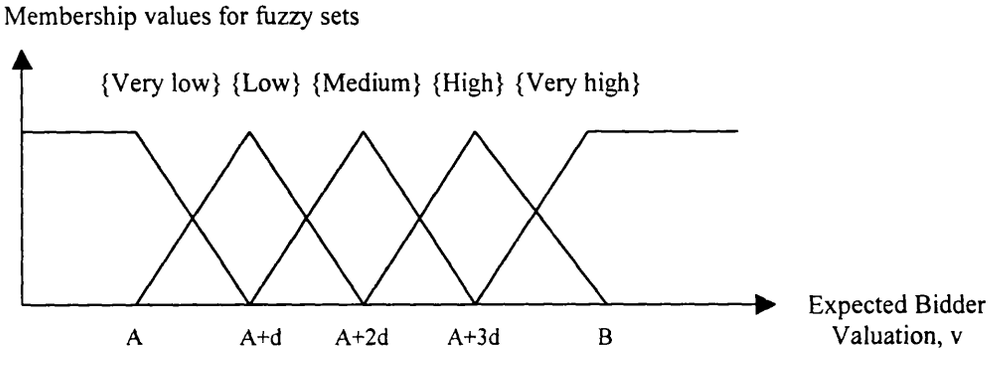


Figure A.1 Fuzzy set partitioning

The following derived equations give the membership values for all the fuzzy sets used to partition 'expected bidder valuation'.

Take the first fuzzy set, {Very Low} where u1(v) is the membership function:

If (v<=A) then u1(v) = 1

If (A<=v<=A+d) then u1(v) = m*v+c (linear equation), where m and c are derived from

Gradient m = -1/d and Intercept c: 1 = -(A)/d+c => c= 1+A/d

So u1(v) = -v/d + 1+ A/d

For the next fuzzy set, {Low} where u2(v) is the membership function:

If (A<=v<= A+d) then u2(v) = m*v+c (linear equation), where m and c are derived from

Gradient m = 1/d and Intercept c: 0 = A/d + c => c =-A/d

So u2(v) =v/d –a/d

If (A+d<=v<=A+2d) then u2(v) =m*v+c (linear equation), where m and c are derived from

Gradient m = -1/d and Intercept c: 1= -(A+d)/d+c => c= 2+A/d

So u2(v) =-v/d + 2 + a/d

This can be done for all the remaining fuzzy sets to derive the equations for their membership functions. This is summarised in the following pseudo code in which p fuzzy sets are used in the partition and FV[i] i=0,1,2,3, ... p-1 is the array of membership values of ith fuzzy set from left to right that partitions v and A<=v<=B then

```
d = (B-A)/(p-1);

if (v <= A) FV[0] = 1;
if (v >= B) FV[p-1] = 1;

for (int i=0;i<(p-1);i++){

        if (v >= (A+i*d) && v < (A+(i+1)*d)){

                FV[i] = -v/d +  (1+i+A/d);
                FV[i+1] = v/d - (i+A/d);
        }
}
```

This forms an array FV of p elements that represents the fuzzy interpretation of a particular value, v in terms of fuzzy membership values for the fuzzy sets used to partition v. This can be repeated for the other two pieces of information to give two additional arrays of membership values. The three arrays are combined with an agent's chromosome to make its decision, see sections 4.2.2 and 6.2.1 for details.

Appendix B - Derivation of Stationary Population Equation

The mathematics for deriving the equation giving the number of children for a ranked pair of agents is as follows. Let the population size be N agents, and let p = 1, ... , N/2 be the ranked pairs, where p=1 is the fittest pair and p=N/2 is the least fit pair. Let C(p) be the number of children had by ranked pair p. Assume C(p) >= C(p+1) for p = 1, ... , N/2-1. Assume C(p) takes a linear form with respect to p and C(1) = 4 and C(N/2) = 0. Let C = m*p + c be the linear equation, where m is the gradient of the equation and c is the constant. To maintain a stationary population of size N the following must hold:

$C(1) = m + c = 4$ (1)

$C(N/2) = 0 = m * N/2 + c$ (2)

Therefore, $m = 4 - c$ or $m = 4 - m * N/2$ and $m = 8/(2- N)$ (3)

Substitute, (3) into (1) to get $c = 4 * N / (N -2)$

So, $C(p) = - 8 * p / (N - 2) + 4 * N / (N -2)$

So the number of children for ranked pair p is C(p) rounded to the nearest integer.

Appendix C - Source Code for the Auctioneer Monitor

```
// ************************
// AUCTIONEER MONITOR CLASS
// ************************

import java.lang.*;

// Auction Monitor controls auction for Bidder agents and one Seller agent

class Monitor{

  // maximum Bidder agents set here
  int MAXBIDDERS = 100;
  int bidderCount;

  // array stores bidders' ID and their bids during the auctions
  String[][] bidLogs = new String[MAXBIDDERS][2];
  int startingPrice;
  int reservePrice;
  int askingBid;
  String auctionOutcome;

  // auction variables
  String user;
  String address;
  String sellerID;
  String lotID;
  int approxPrice;
  String method;
  String start;

  // Seller determines auction method
  // In FPSB and Vickrey auctions
  // Seller sets Reserve Price
  // Bidders set bids
  // Seller reads bids and sets outcome
  // In English and Dutch auctions Seller sets Starting Price
  // Bidders read Starting Price
  // Bidders set answers to the Starting Price
  // Seller reads answers
  // If bidders # > 1 still in auction then another cycle
  // sub-cycle 1: Seller SET ASKING BID
  // sub-cycle 2: Bidders READ ASKING BID
  // sub-cycle 3: Bidders SET REPLIES
  // sub-cycle 4: Seller READ REPLIES
  // All sub-cycles are synchronised

  // Busy waiting is handled by seller/bidders thread waiting
  // BARRIERS set for bid/answer read/write cycles
  // Synchronise all agents together after each cycle

  boolean outcomeSet;
  int bReadAskCount=0;
  int bWriteAnsCount=0;
  int subcycle = 0;

  // status for bidders checking whether they have checked out
  // the auction - only used by bidder
  String status;

  // ***********************************************
  public synchronized int ReadBidderCount(){

    return this.bidderCount;
  }

  // ***********************************************
  // used by Bidders to register with the Seller agent's auction

  public synchronized int Register(String id){
```

```
  bidLogs[bidderCount][0] = id;
  this.bidderCount++;
  return bidderCount-1;
}

// ***********************************************

public synchronized void Initialise(int s){

  if (method.equals("FPSB") || method.equals("Vickrey")) subcycle = 3;
  else subcycle = 1;
  notifyAll();
  System.out.println("Auctioneer Monitor Initialised");
}

// ***********************************************
// these methods used only in English and Dutch auctions
// firstly, seller sets asking bid

public synchronized void SetAskingBid(int b){

  while(subcycle != 1)
    try{
      System.out.println("Seller waiting");
      wait();
    }
    catch(InterruptedException e){;}

  System.out.println("Seller sets asking bid" + Integer.toString(b)); //b is an integer
  askingBid = b;

  // i.e. end of auction
  if (outcomeSet) askingBid = -1;
  subcycle = 2;
  notifyAll();
}

// ***********************************************
// all bidders must read asking bid

public synchronized int ReadAskingBid(){

  while(subcycle != 2)
    try{
      System.out.println("Bidders waiting for Seller to set asking bid");
      wait();
    }
    catch(InterruptedException e){;}

  System.out.println("bidder reads asking bid");
  bReadAskCount++;

  if (bReadAskCount == bidderCount){
    System.out.println("All bidders have read");
    subcycle = 3;
    bReadAskCount=0;
    notifyAll();
  }
  return this.askingBid;
}

// ***********************************************
// All bidders must set bid answers

public synchronized void SetBidAnswer(String s, int bindex){

  while(subcycle != 3)
    try{
      System.out.println("Bidders waiting for Seller to read bid answers");
      wait();
    }
    catch(InterruptedException e){;}

  System.out.println("Bidder sets bid answer" + s);
  System.out.println(Integer.toString(bidderCount));
  bidLogs[bindex][1] = s;
  bWriteAnsCount++;
  if (bWriteAnsCount == bidderCount){
```

```
      subcycle = 4;
      if (method.equals("English") || method.equals("Dutch")) bWriteAnsCount=0;
      System.out.println("All bids set");
      notifyAll();
    }
  }

  // ***********************************************
  // seller must read all bid answers

  public synchronized String[][] ReadBidAnswers(){

    while(subcycle != 4)
      try{
        System.out.println("Seller waiting for bidders to set their bid answers");
        wait();
      }
      catch(InterruptedException e){;}

    System.out.println("Seller reads bid answers");
    subcycle = 1;
    notifyAll();
    return bidLogs; // bindex <= bidderCount
  }

  // ***********************************************

  public synchronized void SetRP(int s){

    System.out.println("Setting Reserve Price");
    reservePrice = s;
  }

  // ***********************************************

  public synchronized void SetOutcome(String o){

    System.out.println("Setting outcome");
    auctionOutcome = o;
    outcomeSet = true;
    notifyAll();
  }

  // ***********************************************

  public synchronized String ReadOutcome(){

    while(!outcomeSet)
      try{
        System.out.println("Bidder waiting");
        wait();
      }
      catch(InterruptedException e){;}

    System.out.println("Bidder read outcome");
    return auctionOutcome;
  }

  // ***********************************************
  // Constructor

  public Monitor(      String m, String a, String se, String l,
                       int price, String me, String st, String s){

    // Information used to reference auction
    this.user = m;
    this.address = a;
    this.sellerID = se;
    this.lotID = l;
    this.approxPrice= price;
    this.method = me;
    this.start = st;
    this.status = s;
    this.bidderCount = 0;
    outcomeSet = false;
  }
}
```

Appendix D – Source Code for the Agent Router

```
// ****************************
// AGENT ROUTER PERL CGI SCRIPT
// ****************************

#!/usr/local/bin/perl -Tw
require 5.003;
use strict;

# An Agent Router for the Intelligent Auction System
# Windows 95 and NT version

# Mobile User Support:
# Login (Name, Password, IP:Port and Filter)
# if incorrect resubmit
# if OK then this info is used to update IP, Port and Filter

#all variables
my ($length,$allfile,$from,$fromaddress,$dataType,$to,$toaddress);
my ($data1,$data2,$data3,$data4,$data5,$data6,$reply,$remote);
my ($remotehome1,$remotehome2,$user,$port,$iaddr,$paddr);
my ($proto,$info,$env_var,%list,$name,$foundAddr,$foundFilter,$path,$line);
my ($sport,$sremote,$thisScript,$OK,$destnDate,$pid);
my ($USreply,$filter,$file1,$file2,$this,$that,$type,$sockaddr,$len,
my ($thataddr,$child,$aliases,$thisaddr,$AF_INET,$SOCK_STREAM,$hostname,$them);
my ($foundUser,$pass,$logon,$el,$bl,$d,$newline,$blank,$i);
my ($MAXLENGTH,$piece,$loop);

# $USreply refers to a User accepting
# an Agent from a filtered source
# Filter by default
$USreply = 'REJECT';

# FOR UNIX SYSTEMS
# note the permissions are in decimal
# note the process spawned from httpd is 'nobody'
# and has few permissions
# need to set this script with suid then this script
# can form a socket for interprocess comms
# $thisScript = 'c:\website\cgi-shl\router.pl';
# chmod only for UNIX systems
# chmod 2559, $thisScript;

##########################################################

$length = $ENV{'CONTENT_LENGTH'};
$allfile = "";
binmode STDIN;
sysread STDIN,$allfile,$length;

# This CGI script will route and check security on messages and
# agents. The CGI will not do any parsing. Routing has two aspects to it:
# The User and the unique manager address
# Since a User can't run more than one Manager, security
# check is made on someone trying to log in twice under one name

# Tainted Perl variables become tainted so use ENV directly
# note: the Username.txt should hold all the current information about
# Users, Managers and Displays etc so that agent(s) can be routed.

# Protocol delimiter is:
# $delimiter = ":::";

# from implies where the agent originated from
($from,$fromaddress,$dataType,$to,$toaddress,$data1,$data2,$data3,$data4,$data5,$data6) =
split(/:::/,$allfile,11);

# default data type
if ($dataType eq ''){
  $dataType = "PROBLEM";
}
```

```
##################################################
# CGI requests:-
$path = 'C:\IAS\Configs\Username.txt';
# name:::host:::port:::password:::filter:::logonstatus
# beginline and endline byte markers, $bl, $el
$bl = 0;
$el = 0;
$MAXLENGTH = 80;

# open for reading and writing
open (USERS,"+<".$path);

if ($dataType eq 'LOGON'){

  # check user's name and password
  # data3 is password, data1 is Manager IP, data2 is port
  # data4 is filter (yes/no) and data 5,6 are blank

  $foundUser = 'false';

  LOOP: for(;;){

    $bl = tell USERS;
    $line = <USERS>;
    $el = tell USERS;

    chomp $line;
    ($name,$sremote,$sport,$pass,$filter,$logon) = split(/:::/,$line,6);

    if ($to eq $name && $data3 eq $pass){

      # found user's name and password
      $foundUser = 'true';
      # security check to allow only one login per User

      if ((substr $logon,0,3) eq 'OFF'){

        # update Username.txt with new host, port and filter
        # write line back to USERS file
        # check reply is OK for Origin/Proxy servers
        # length of line must be fixed and set to MAX = 200 bytes
        $length = $MAXLENGTH - $el + $bl;
        seek USERS,$bl,0;
        # add 150 - $length bytes to $newline;
        $blank = '';

        for($i = 0; $i < $length; $i++){
          $blank = $blank.'*';
        }

        $d = ":::";
        $newline = $name.$d.$data1.$d.$data2.$d.$data3.$d.$data4.$d.'ON'.$blank;
        print USERS $newline;
        print "Content-type: text/plain"."\n\n";
        print "User Cleared";
      }
      else {
        print "Content-type: text/plain"."\n\n";
        print "User Violation";
      }

      close(USERS);
      exit(0);
    }

    last LOOP if (eof USERS);
  }

  if ($foundUser eq 'false'){

    $reply = "User or Password Invalid?";
    print "Content-type: text/plain"."\n\n";
    print "Incorrect Details";
    close(USERS);
    exit(0);
  }
}
else {
```

```
# Check the requesting agent's destination is valid
# if it is not they reply with REQUESTED DESTINATION DOES NOT EXIST?
# if it is then check IAS destination's filter
# if filter is off then agent can pass through
# if filter on then ask destination IAS if agent can pass through
$foundUser = 'false';

# filter or not (not per User but all or nothing)

LOOP: for(;;){

  $bl = tell USERS;
  $line = <USERS>;
  $el = tell USERS;
  chomp $line;
  ($name,$sremote,$sport,$pass,$filter,$logon) = split(/:::/,$line,6);

  # note that IP names can have - in them
  # tainted - used for sockets so need subexpression
  $sremote=~/(([-\w]+)(\.[-\w]+)*)/;
  $remote = $1;
  $sport=~/([\d]+)/;
  $port = $1;

  if ($filter eq 'ON' || $filter eq 'OFF'){
    $foundFilter = 'true';
  }

  if ($to eq $name){
    # found destination's name, IP, port and filter
    # if filter is on then must ask whether they
    # want to accept the agent (for every agent)
    $foundUser = 'true';
    last LOOP;
  }

  last LOOP if (eof USERS);
}

if ($foundUser eq 'false'){

  $reply = "REQUESTED DESTINATION DOES NOT EXIST?";
  print "Content-type: text/plain\n\n";
  print "INCORRECT ADDRESS";
  exit(0);
}

if ($dataType eq 'LOGOFF'){

  # update Username.txt file
  $length = $MAXLENGTH - $el + $bl;
  seek USERS,$bl,0;
  # add 150 - $length bytes to $newline;
  $blank = '';

    for($i=0; $i < $length; $i++) $blank = $blank.'*';

    $d = ":::";
    $newline = $name.$d.$sremote.$d.$sport.$d.$pass.$d.$filter.$d.'OFF'.$blank;
    print USERS $newline;
    print "Content-type: text/plain"."\n\n";
    print "Logged Off";
    exit(0);
  }

  ##########################################################
  # make sockets, bind and connect them to Manager's server
  # this forms a comms link betweeen CGI and the Manager

  $them = $remote;
  $them = 'localhost' unless $them;
  sub dokill { kill 9,$child if $child; }
  use Socket;
  $sockaddr = 'S n a4 x8';
  chop($hostname = `hostname`);
  ($name, $aliases, $proto) = getprotobyname('tcp');
  ($name, $aliases, $port) = getservbyname($port, 'tcp') unless $port =~ /^\d+$/;
```

```
($name, $aliases, $type, $len, $thisaddr) = gethostbyname($hostname);
($name, $aliases, $type, $len, $thataddr) = gethostbyname($them);
$this = pack($sockaddr, AF_INET, 0, $thisaddr);
$that = pack($sockaddr, AF_INET, $port, $thataddr);
socket(SOCK, PF_INET, SOCK_STREAM, $proto) || die "socket: $!";
bind(SOCK, $this) || die "bind: $!";
connect(SOCK, $that) || die "connect: failed";

#########################################################

# protocol for CGI - Manager communications

if ($dataType eq 'TIME'){

  # NOT used in current IAS prototype
  # To visit auctions at the right time (GMT/LOCAL)

  send(SOCK,$from."\n",0);
  send(SOCK,"TIME\n",0);

  # get the date and send it back to originator
  # winsock cannot fork: because no threads or forking
  # has to be done independently - priority reader/writer etc.

  chomp($destnDate = <SOCK>);
  chomp($OK = <SOCK>);

  if ($OK eq 'DATE TIME TRANSMITTED'){

    print "Content-type: text/plain\n\n";
    print $destnDate."\n";
    print $OK;
  }
  else {
    print "Content-type: text/plain\n\n";
    print "MANAGER DOWN OR UNREACHEABLE";
  }
}
elsif ($dataType eq 'BIDDER'){

  send(SOCK,$from."\n",0);
  send(SOCK,"BIDDER"."\n",0);

  $length = length $data1;
  send(SOCK,$length."\n",0);
  # data1 is the agent script

  syswrite SOCK, $data1, $length;

  print "Content-type: text/plain\n\n";
  print "AGENT HAS LEFT";
}
elsif ($dataType eq 'UPDATE'){

  # anything that must happen remotely (IAS) if agent .... departs ,arrives
  send(SOCK,$from."\n",0);
  send(SOCK,"UPDATE"."\n",0);

  # data1 is details and data2 is ID
  send(SOCK,$data1."\n",0);
  send(SOCK,$data2."\n",0);

  print "Content-type: text/plain"."\n\n";
  print "UPDATED";
}
elsif ($dataType eq 'LOG'){

  send(SOCK,$from."\n",0);
  send(SOCK,"LOG"."\n",0);

  # data1 is details and data2 is ID
  send(SOCK,$data1."\n",0);
  send(SOCK,$data2."\n",0);

  print "Content-type: text/plain","\n\n";
  print "LOGGED";
}
elsif ($dataType eq 'ACCEPTANCE'){
```

```
        if ($to eq $from && $toaddress eq $fromaddress){

          send(SOCK,$from."\n",0);
          send(SOCK,"ACCEPTED"."\n",0);

          print "Content-type: text/plain\n\n";
          print "ACCEPTED";
        }
        if ($foundFilter eq 'true' && $filter eq 'OFF'){

          send(SOCK,$from."\n",0);
          send(SOCK,"ACCEPTED"."\n",0);

          print "Content-type: text/plain\n\n";
          print "ACCEPTED";
        }
        elsif ($foundFilter eq 'false' || $filter eq 'ON'){

          # must ask User if they want to accept agent from source
          # data1 is the source's User Name
          # data2 is Machine Address
          # data3 is Agent's ID/Name

          send(SOCK,$from."\n",0);
          send(SOCK,"CHECK SOURCE"."\n",0);
          send(SOCK,$data1."\n",0);
          send(SOCK,$data2."\n",0);
          send(SOCK,$data3."\n",0);

          chomp($USreply = <SOCK>);

          if ($USreply eq 'ACCEPTED'){

            send(SOCK,$from."\n",0);
            send(SOCK,"ACCEPTED"."\n",0);

            print "Content-type: text/plain\n\n";
            print "ACCEPTED";
          }
          elsif ($USreply eq 'REJECTED'){

          send(SOCK,$from."\n",0);
          send(SOCK,"REJECTED"."\n",0);

          print "Content-type: text/plain\n\n";
          print "REJECTED";
        }
      }
    }
    elsif ($dataType eq 'PROBLEM'){

    # default is a problem...

    send(SOCK,$from."\n",0);
    send(SOCK,"PROBLEM"."\n",0);

    # data1 is problem message
    send(SOCK,$data1."\n",0);

    print "Content-type: text/plain\n\n";
    print "OK";
  }
}

close (SOCK);
exit(0);
```

Appendix E – Partial Source Code for the Agent Client/Server

This appendix shows the main structure of the agent client (Manager class), agent server (Listener and Gateway classes) and their constructors. In addition, it shows the method used to post agents to remote agent routers using HTTP.

```
// *************
// MANAGER CLASS
// *************
import java.awt.*;
import java.net.*;
import java.io.*;
import java.util.*;
import java.lang.*;

// IAS is a multi-threaded client-server system.
// The five main types of thread are:
// Interrupt GUI(s), Clock, Bidder agents, Seller agents and Listener (server).

// The poster method is used to post messages and agent(s) to web server agent router(s)
// All start up information is held in the user's Config.ini file and Username.txt

public class Manager extends Main_GUI{

  // GUI(s) for login & Viewing, Saving & Creating Agent(s)
  Login_GUI f1;
  Users_GUI f2;
  SetupSeller_GUI Sr;
  SetupBidder_GUI Br;
  View_GUI f4;
  Load Lo;
  Save_GUI f6;
  Simulator_GUI f7;
  Logout L;
  ViewAuc_GUI f8;

  // IAS Manager ID
  String User;
  String IAS_IP;
  String IAS_Port;
  boolean on = false;

  // Web server's CGI Address: http://domain name/cgi rel path for proxies
  // or /cgi-bin/script for origin servers with Host: field complete
  String webserverDomainName;
  String CGI_Address;
  String serverAccess;

  // Proxy server host or local (if no proxy) and Telnet port for HTTP
  // Normally proxy's port is 8000 for firewall and 80 for local web server
  String host;
  String port;

  Thread clock;

  // vector of agent threads
  Vector sellerAgents = new Vector();
  Vector bidderAgents = new Vector();

  // array of scripts: loaded or completed
  String currentAgentID;
  String currentScript;
  int MAXPOOL = 50;
  int NumInPool = 0;
  String pooledAgents[] = new String[MAXPOOL];
  int idindex = 0;
  String bidderIDs[] = new String[MAXPOOL];
```

```
// this holds all log info for the agents
String LOG = new String("**** LOG ****\n");

// The Listener always runs, listening for incoming Agent(s) & messages
Listener listener;

// ALL refs are lost when Manager is off. A file could be
// used in the Monitor constructor to read a file of auction
// refs and build up the auction vector. This would allow
// agents to be stopped, saved and reloaded and thus would be able
// to pick up from where they left off
Vector monitors = new Vector();
Vector auctionAddresses = new Vector();

public Manager(String title){

  super(title);
  IAS_Status.setText("Security Check: Please Login");
  // setting the clock's thread
  clock = new Clock(this);
  clock.setPriority(4);
  clock.start();

  // IAS is disabled until the correct User name, password is supplied
  // each local IAS will have a unique server listening on the defined port
  // disable buttons
  login.enable();
  logout.disable();
  // other buttons

}

// ***********************************************

public static void main(String args[]) throws IOException{

  Frame f = new Manager(" INTERNET AUCTION SYSTEM");
  f.setBackground(Color.lightGray);
  f.pack();
  f.show();
}

// ***********************************************
//    Functionality for the Manager's GUI
// ***********************************************
// Rest of Code...


// ***********************************************
// Poster method to post agents/messages using HTTP
// ***********************************************

public synchronized String IAS_Poster( String fromSender,String fromAddress,
                                       String messageType, String toUser,
                                       String toAddress,
                                       String info1, String info2, String info3,
                                       String info4, String info5, String info6) {

  // info is either an agent, security info or a message
  // protocol will be determined by Agent and message formats
  // The agent will use this to transport itself or send messages

  while(on)
    try{ wait(); }
    catch(InterruptedException e){;}

  on = true;
  String delimiter = new String("::::");
  String allData;
  Socket s;

  // Security: info1 is password, info2 is agent address and info3 is Manager port
  // Message or Agent: info1 the Agent or info 1-6 are extra pieces of information
  allData = new String( fromSender + delimiter + fromAddress + delimiter +
                        messageType + delimiter + toUser + delimiter + toAddress +
                        delimiter + info1 + delimiter + info2 + delimiter + info3 +
                        delimiter + info4 + delimiter + info5 + delimiter + info6);
```

```
    try{

      // want to send data or info to toAddress (web server) and not local web server
      host = toAddress;
      s = new Socket(host,Integer.parseInt(port));

      DataInputStream sin = new DataInputStream(s.getInputStream());
      // note: PrintStream (ASCII text) : DataOuptuStream (Binary)
      DataOutputStream sout = new DataOutputStream(s.getOutputStream());

      //IAS_Status.setText("Connected to " + s +s.getInetAddress()+":"+ s.getPort());
      String HTTP_Request;
      String HTTP_Reply;
      String CRLF  = "\r\n";
      StringBuffer buffer = new StringBuffer("");
      int data_Size = allData.length();

      // Username and web server domain name is supplied by agent
      // if in doubt look in their Username.txt file
      // path is same for all agent routers in IAS
      // Origin and Proxy server requests
      HTTP_Request = new String( "POST "+ CGI_Address + " HTTP/1.0" + CRLF +
                                 "Host: " + toAddress + CRLF + "Content-length: " +
                                 Integer.toString(data_Size) + "\n\n" + allData);

      int len = HTTP_Request.length();
      byte Bytes[] = new byte[len];
      HTTP_Request.getBytes(0,len,Bytes,0);
      sout.write(Bytes,0,len);

      //System.out.println(HTTP_Request);
      IAS_Status.setText("Data being sent to its Destination");
      String line = null;

      try{
        line = sin.readLine();
      }
      catch (EOFException e){;}

      for(;;){
        try{
          line = sin.readLine();
          if (line.equals("")) break;
        }
        catch (EOFException e){ break;}
      }

      for(;;){
        try{
          line = sin.readLine();
          if (line == null) break;
          buffer.append(line+"\n");
        }
        catch (EOFException e){break;}
      }

      HTTP_Reply = buffer.toString();
      System.out.println(HTTP_Reply);
      IAS_Status.setText("DATA ACCEPTED");

      s.close();
      on = false;
      notifyAll();
      return(HTTP_Reply.trim());
    }
    catch (IOException e){

      IAS_Status.setText("Input Output Problem");
      on = false;
      notifyAll();
      return("IO Problem");
    }
  }
}
```

```
// ***************************
// IAS (LISTENER) SERVER CLASS
// ***************************

// Listens for incoming agents from local
// or remote agent routers on a well-defined port.
// The agent router must form a socket connection
// with the listener's port to pass the agent or message.

import java.awt.*;
import java.net.*;
import java.io.*;
import java.util.*;
import java.lang.*;

class Listener extends Thread {

  // The IAS server must always be ready for
  // incoming messages and agent(s)
  Manager ias;
  int port; // unique port on computer
  protected ServerSocket listen_socket;
  String type, message, Agent;
  DataInputStream in;

  public Listener(Manager i){
    super("IAS Listener");
    this.ias = i;
    this.port = Integer.parseInt(ias.IAS_Port);
  }

  public void run(){

    try{

      listen_socket = new ServerSocket(port);
      ias.IAS_Status.setText("Listening for Agent(s)");
      // Agent or message
      // protocol determined by agent and message formats
      in = null;

      while(true){

        try{
          Thread.currentThread().sleep(1000);
        }
        catch(InterruptedException e){;}

        Socket CGI_socket = listen_socket.accept();
        Gateway c = new Gateway(CGI_socket,ias);
      }
    }
    catch (IOException e){
      ias.IAS_Status.setText("Problems with Gateway");
    }
    finally{
      try{
        in.close();
        listen_socket.close();
      }
      catch(IOException e){;}
    }
  }
}
```

```
// *****************
// IAS GATEWAY CLASS
// *****************
// This class accepts a socket connection with an agent router
// to form a gateway between the web server and the agent server.

import java.awt.*;
import java.net.*;
import java.io.*;
import java.util.*;
import java.lang.*;

class Gateway extends Thread{

  Manager ias;
  Socket CGI = null;
  protected DataInputStream in;
  protected PrintStream out;
  String US_from;
  String US_title;
  String US_IP;
  String US_port;

  public void Update(String ID,String details){

    // Update Bidder List
    int index = -1;
    int bidderCount = ias.bidders.countItems();

    for(int i=0;i<bidderCount;i++){
      if (ias.bidders.getItem(i).startsWith(ID)){
        index = i;
        break;
      }
    }
    if (index == -1) ias.bidders.addItem(details);
    else{
      try{
        ias.bidders.replaceItem(details,index);
      }
      catch(ArrayIndexOutOfBoundsException e){;}
    }
  }

// ********************************************

  public Gateway(Socket CGI_socket, Manager i){

    this.ias = i;
    CGI = CGI_socket;
    try{
      in = new DataInputStream(CGI.getInputStream());
      out = new PrintStream(CGI.getOutputStream());
    }
    catch(IOException e){;
      ias.IAS_Status.setText("Problems with Streams");
    }
    this.start();
  }

  public void run(){

    String from;
    String line;
    StringBuffer data = new StringBuffer("");
    String dataType;

    // protocol is: from:dataType:data (Agent or Message)
    // size of Agent will have to be dealt with

    try{
      // asynchronous comms sockets
      // using threads and forking processes
      from = in.readLine();
      dataType = in.readLine();
      ias.IAS_Status.setText("Message From: " + from + " of Type: " + dataType);
```

```
    if (dataType.equals("TIME")){

      Date d = new Date();
      out.println(d.toLocaleString());
      out.println("DATE TIME TRANSMITTED");
    }
    else if ((dataType.trim()).equals("BIDDER")){

      int len = Integer.parseInt(in.readLine());
      byte Bytes[] = new byte[len];
      in.read(Bytes,0,len);
      String agent = new String(Bytes,0);
      String BidderID = ias.ID(agent); // extract ID
      ias.IAS_Status.setText("New Bidder Agent Arrival");
      System.out.println("ID=" + BidderID);
      Bidder bagent = new Bidder(ias,BidderID,agent,from);
      ias.bidderAgents.addElement(bagent);
      bagent.start();
    }
    else if ((dataType.trim()).equals("LOG")){

      // Message one
      String details = new String(in.readLine());
      // need Bidder ID
      // extract bidder_ID and details from the cgi data
      String Bidder_ID = new String(in.readLine());
      ias.LOG = ias.LOG + Bidder_ID + " " + details + "\n";
    }
    else if ((dataType.trim()).equals("UPDATE")){

      // Message two
      String details = new String(in.readLine());
      // need Bidder ID
      // extract bidder_ID and details from the cgi data
      String Bidder_ID = new String(in.readLine());
      // update the bidder list
      Update(Bidder_ID,details);
    }
    else if (dataType.equals("UNKNOWN SOURCE")){

      // display who the Agent is from and ask whether the Agent is accepted
      // the name, Agent title, machine and port is read via the socket
      US_from = from;
      US_title = in.readLine();
      US_IP = in.readLine();
      US_port = in.readLine();

      AgentFilter_GUI f1 = new AgentFilter_GUI("AGENT ACCEPT-REJECT FORM",this,ias);
      f1.pack();
      f1.show();
    }
    else if ((dataType.trim()).equals("ACCEPTED")){
      ias.IAS_Status.setText("Bidder Agent From " + from + " Has Been Accepted");
    }
    else if (dataType.equals("REJECTED")){
      ias.IAS_Status.setText("Bidder Agent From " + from + " Has Been Rejected");
    }
    else{
      ias.IAS_Status.setText("Destination Not Reachable");
      // pass message on to the correct Bidder Agent via thread number
      // i.e. update IA Status part of the Agent's script!
      // if a problem with the next IAS then this is passed to the Agent
      // via the <IA STATUS> tag
    }
  }
  catch (IOException e){
    ias.IAS_Status.setText("Problems with Gateway Socket");
  }
  finally{
    try{
      CGI.close();
      in.close();
      out.close();
    }
    catch(IOException e){;}
  }
}
```

Appendix F – Partial Source Code for SAML and BAML Tag Checkers

This appendix shows partial source code for SAML and BAML tag checkers. The main methods check that the agent script has the correct tags, that they are in the right order and that they mark up data of the right type. Since, both Bidder and Seller agent scripts are checked in the same way only code for the SAML tag checker is given.

```
// ******************************
// Tag Checkers for Agent Scripts
// ******************************
// The code below demonstrates how the tags are checked
// for Seller agents written in SAML

public class SAML_ParserI{

  // Input script is ss and outputs are error messages
  // If there are no errors then it should run correctly
  // Though some inter-tag semantics are not checked
  // The agent can then be interpreted by the interpreter
  // errors are appended to this buffer

  public StringBuffer error;

    // constructor
    public SAML_ParserI(String ss){

    // The script is NOT broken into tokens and checked against the lexicon
    // with symbol tables etc. It is scanned for markup tags - checking that they
    // are present, in proportion and in the correct order.
    // Semantic checking looks for the data between the markup tags
    // This compares to checking a markup script against an SGML DTD
    // formalised using BNF.
    // These are static methods because they are used by the class and
    // not a particular instance (object of the class)

    error = new StringBuffer();

    // Lexical Parse - check tag types and numbers
    SAML_Lexical(ss);

    // Syntax Parse - check tag order
    SAML_Syntax(ss);

    // Semantic Parse - check data types
    SAML_Semantic(ss);
  }

  // ***********************************************

  public  void Tag_Order(String s, String starttag,String endtag){

    // check that <XXX> and its end <\XXX> are in order
    // or that <XXX> should come before <YYY>

    boolean errorB = false;

    if (s.indexOf(starttag,0) >= s.indexOf(endtag,0)){

      errorB = true;
      // tags wrong order
      error.append("Error: " + starttag+ " " + endtag + " Tags incorrect order\n");
    }
  }
```

```
public  void Tag_Sandwich(String s,String tag,String brackettag){

  // check that <XXX> and </XXX> is sandwiched by <YYY> and </YYY>
  boolean errorB = false;

  if (s.indexOf("<"+tag+">") <= s.indexOf("<"+brackettag+">") ||
      s.indexOf("<"+tag+">") >= s.indexOf("</"+brackettag+">")){

    errorB = true;
    // <XXX> tag is not sandwiched by <YYY> </YYY>
    error.append("Error: "+"<"+tag+">"+" is not sandwiched by"+"<"+brackettag+">\n");
  }

  if (s.indexOf("</"+tag+">") <= s.indexOf("<"+brackettag+">") ||
      s.indexOf("</"+tag+">") >= s.indexOf("</"+brackettag+">")){

    errorB = true;
    // </XXX> tag is not sandwiched by <YYY> </YYY>
    error.append("Error: "+"</"+tag+">"+" is not sandwiched by"+"<"+brackettag+">\n");
  }
}

// ***********************************************

public  void Tag_Check(String s,String tag1,String tag2,int n,boolean proportion){

  // Three main checks:
  // (1) 1 of tag1 and 1 of tag2 only (proportion is yes)
  // (2) n>1 of tag1 and n>1 of tag2  (proportion is no)
  // (3) 1 of tag1 and n>=1 of tag2   (proportion is no)
  // implies one whilst n = 2 implies 2 or more

  boolean errorB = false;

  int pos1 = 0; int from1 = 0; int pos2  = 0; int from2 = 0;

  // keeping count
  int c1 = 0; int c2 = 0;

  while(pos1 != -1 && pos2 != -1){
    if ((pos1 = s.indexOf(tag1,from1)) != -1) c1++;
    if ((pos2 = s.indexOf(tag2,from2)) != -1) c2++;
    from1 = pos1+1;
    from2 = pos2+1;
  }

  // check for tag numbers and tags are present
  if (c1==0){
    error.append("Error: " + tag1 + " is missing\n");
    errorB = true; // tag1 missing
  }

  if (c2 ==0){
    error.append("Error: " +tag2 + " is missing\n");
    errorB = true; // tag2 missing
  }

  // check 1:1
  if (proportion && n==1){

    if (c1 > 1){
      error.append("Error: "+tag1 + " is Repeated\n");
      errorB = true; // another tag
    }

    if (c2 > 1){
      error.append("Error: "+tag2 + " is Repeated\n");
      errorB = true; // another tag
    }

  }
  else if (proportion && n==2){

    // check (n:n) n of tag1 and n of tag2
    if (c2 > c1){
      error.append("Error: Tag Ratio Mismatch: more of "+ tag2 + " than " + tag1+"\n");
      errorB = true; // tag ratio mismatch
    }
```

```
    else if (c1 > c2){
      error.append("Error: Tag Ratio Mismatch:  more of "+ tag1 +" than " + tag2+"\n");
      errorB = true; // tag ratio mismatch
    }

  }
  else if (!proportion){

    //check 1:n
    if (c1 > 1){
      error.append("Error: " + tag1 + " is Repeated\n");
      errorB = true; // another tag
    }
  }
}

// ***********************************************

public  void MetaTags_Check(String s,String meta, String list[]){

 // this method checks that the meta tags contain the correct tags
 String metatagstart = new String("<"+meta+">");
 String metatagend = new String("</"+meta+">");
 String datatag;
 String test;

 int number = list.length;
 // extract characters between <metatag> and <xxx>
 int begin = s.indexOf(metatagstart);
 int end   = s.indexOf(metatagend);

 if (begin == -1 || end == -1){
    error.append("Error: metatag " + meta + " doesnot exist in script\n");
    return;
 }

 try{
   test = s.substring(begin+metatagstart.length(),end);
 }
 catch(IndexOutOfBoundsException e){
   error.append("Error: with meta-tag <"+meta+">\n");
   return;
 }

 // strip spaces from front and back of string
 int bs,be; int l = test.length();

 for(bs=0;bs<l;bs++)
   if (!(test.charAt(bs)=='\n' || test.charAt(bs)==32 || test.charAt(bs)=='\t')) break;

 for(be=1;be<=l;be++)
   if (!(test.charAt(l-be)==32||test.charAt(l-be)=='\n'||test.charAt(l-be)=='\t')break;

 try{
   test = s.substring(begin+metatagstart.length()+bs,end-be+1);
 }
 catch(IndexOutOfBoundsException e){
   error.append("Error: with meta-tag <"+meta+">\n");
   return;
 }

 //System.out.println(Integer.toString(bs+be));
 //System.out.println(test);
 boolean found = false;

 for(int i=0;i<number;i++){

   datatag = list[i];
   //System.out.println(datatag);

   if (test.startsWith("<"+datatag+">")){
     // no error
     found = true;
     break;
   }
 }

 if (!found) error.append("Error:"+metatagstart + " start data tag missing\n"){
```

```
    found = false;

    for (int i=0;i<number;i++){
      datatag = list[i];

      if (test.endsWith("</"+datatag+">")){
        // no error
        found = true;
        break;
      }
    }

    if (!found) error.append("Error: "+metatagend+" must precede by a end data tag\n");
}

// ********************************************

public  void DataType_Check(String s,String tag,String type, boolean blankAllowed){

    //This method checks that each tag marks up data of the correct type
    int start = s.indexOf("<"+tag+">");
    int end = s.indexOf("</"+tag+">");

    if ( start == -1 || end == -1){
      error.append("Error: <" + tag + "> doesnot exist in script\n");
      return;
    }

    String data=null;

    try{
      data = s.substring(start+("<"+tag+">").length(),end);
    }
    catch(IndexOutOfBoundsException e){
      error.append("Error: with data for tag <"+tag+">\n");
    }

    // if a blank string is allowed check
    if (blankAllowed && data.equals("")) return;

    // Data Types between tags
    // ALPHANUM
    // DATETIME
    // ENUMSTATE, ENUMAUCTYPE
    // POSINT, INT, PARTITION
    // CHROMO string of digits

    if (type.equals("ALPHANUM")){

      if (data.indexOf(">")!=-1 && data.indexOf(">")!=-1 && data.indexOf("/") != -1){
         error.append("Error " + data + " contains markup metacharacters : {<,>,/}\n");
      }
    }
    else if (type.equals("DATETIME")){

      try{
        Date d = new Date(data);
      }
      catch(IllegalArgumentException e){
        error.append("Error : date-time " + data +" format exception\n");
      }
    }
    else if (type.equals("ENUMSTATE")){

      if (!data.equals("STARTING") && !data.equals("WAITING")&& !data.equals("PREPARING")
           && !data.equals("AUCTIONING") && !data.equals("COMPLETING")){

        error.append("Error:"+"<"+tag+">"+" & "+"</"+tag+">"+ " is not a state\n");
      }
    }
    else if (type.equals("ENUMAUCMETHOD")){

      // rest of tag value defintions
    }

}
```

```
// ***********************************************

public  void SAML_Lexical(String source){

  // the source string is lexically parsed by this method
  // any errors are written to file
  // checks are: all tags are present and in proportion
  //             correct use of <> </>
  //             use of <> etc in script text or anywhere else
  String s = new String(source);
  boolean no = false;
  boolean yes = true;

  Tag_Check(s,"<SAML>","</SAML>",1,yes);
  Tag_Check(s,"<ID>","<ADDRESS>",1,yes);
  Tag_Check(s,"<ID>","<CREATOR>",1,yes);
  Tag_Check(s,"<ID>","<DATETIME>",1,yes);
  Tag_Check(s,"<ID>","</ID>",1,yes);


  // Rest of tag checks ...


  Tag_Check(s,"<HIGH BID>","</HIGH BID>",1,yes);
  Tag_Check(s,"<SECOND HIGH BID>","</SECOND HIGH BID>",1,yes);
  Tag_Check(s,"<LOW BID>","</LOW BID>",1,yes);
  return;
}

// ***********************************************

public  void SAML_Syntax(String source){

  // the source string is syntactically parsed by this method
  // any errors are written to file
  // checks are: correct order of tags <X> </X>
  //             matching <X> </X>
  //             only one <X> per <Y> etc
  //             <SAML> tag must come first and </SAML> last
  String s = new String(source);
  boolean no = false;
  boolean yes = true;

  if (s.trim().indexOf("<SAML>") != 0){
    // error
    error.append("Error: <SAML> is not the first tag\n");
  }
  if (!s.trim().endsWith("</SAML>")){
    // error
    error.append("Error: </SAML> is not the last tag\n");
  }

  // all the sub mark-up parts of the script can come in any order
  // ID SCHEDULE AUCTION ASSUMPTIONS FUZZY PARTITION GENOTYPE MEMORY

  Tag_Order(s,"<SAML>","</SAML>");
  Tag_Sandwich(s,"ADDRESS","ID");
  Tag_Sandwich(s,"CREATOR","ID");
  Tag_Sandwich(s,"DATETIME","ID");
  Tag_Order(s,"<ADDRESS>","</ADDRESS");
  Tag_Order(s,"<CREATOR>","</CREATOR");
  Tag_Order(s,"<DATETIME>","</DATETIME");


  // rest of tag checks


  Tag_Order(s,"<AUCTION ASSUMPTIONS>","</AUCTION ASSUMPTIONS");
  Tag_Sandwich(s,"LOG BIDDER","MEMORY");
  Tag_Sandwich(s,"BIDDER ID & BID","LOG BIDDER");

  return;
}
```

```
  // ***********************************************

  public  void SAML_Semantic(String source){

    String s = source;
    // the source string is semantically parsed by this method
    // any errors are written to file
    // checks are:  data type within tags <X> </X> i.e no <> except <DOCUMENT>
    //              Go through every <X> and </X> and check semantics
    // two types of tag : (1) Datatags that have associated data
    //                    (2) Metatags that group these tags
    // for type1 tags -
    // e.g. check datatype between <ADDRESS> and </ADDRESS> is ALPHANUM
    DataType_Check(s,"ADDRESS","ALPHANUM",false);
    DataType_Check(s,"CREATOR","ALPHANUM",false);
    DataType_Check(s,"DATETIME","DATETIME",false);

    // rest of tag checks

    // for type2 tags -
    // e.g. check <SAML> must have either <ID>, <AUCTION ASSUMPTIONS>, ... after it
    //      check </SAML> must have either </ID>, </AUCTION ASSUMPTIONS>, ... before it

    String SELLER[] = { "ID",
                        "AUCTION ASSUMPTIONS",
                        "FUZZY PARTITION",
                        "GENOTYPE",
                        "MEMORY"};

    String ID[] = {     "CREATOR",
                        "ADDRESS",
                        "DATETIME"};

    String AUC[] = {    "LOT ID",
                        "AUCTION TYPE",
                        "AUCTION METHOD",
                        "START",
                        "DELTABID",
                        // rest of tags ... };

    // rest of tag checks ...

    // Meta tag checks
    MetaTags_Check(s,"SAML",SELLER);
    MetaTags_Check(s,"ID",ID);
    MetaTags_Check(s,"AUCTION ASSUMPTIONS",AUC);
    MetaTags_Check(s,"FUZZY PARTITION",ENV);
    MetaTags_Check(s,"GENOTYPE",GENO);
    MetaTags_Check(s,"MEMORY",MEM);
    MetaTags_Check(s,"LOG BIDDER",LOG);
    // need to check for schedule inconsistencies
    return;
  }
}
```

Appendix G – Partial Source Code for Bidder and Seller Agent Interpreters

This appendix shows partial source code for the Seller and Bidder agent interpreters. Since they are long programs, only the main structure is shown with the main methods. Methods that are used in the both Seller and Bidder agent interpreters are not repeated in the Bidder agent interpreter code listing. In addition, methods used by agents to make decisions are listed in Appendix H for the simulated agents since identical methods are used.

```
// ******************************
// SELLER AGENT INTERPETER CLASS
// ******************************

class Seller extends Thread{

  // Seller script is interpreted so that it behaves as an automaton
  // Seller states are WAITING, PREPARING, AUCTIONING, COMPLETING
  // Events are time, bidder numbers and auction rules
  // Actions are updating IAS screen, conducting auction (read/write to scripts)
  // ***********************************************
  // Essential parameters for thread
  // SELLER SCRIPT is Self-modifiable hence a StringBuffer is used

  StringBuffer Script;
  String thread_ID;
  Manager ias;
  int p1,p2,p3; // length of fuzzy vectors

  // ***********************************************
  // Seller makes one or two decisions (~ Auction Method) - RP/SP
  // chromo x fv - continuous range output PV[min,V] or CV[min,max]
  // Auction environment is described by these three fuzzy vectors
  // FV1 is belief abt bidders average valuation
  // FV2 id belief abt bidders valuation range
  // FV3 is belief abt bidders scaledown (CV only - zero if PV)
  // Each decision chromosome consists of genes
  // Each gene can be value from the range [0,N] i.e N+1 decision partition
  // RP Decision = chromo x FV = PV[V,max], CV[min,max] all auctions methods
  // SP Decision = chromo x FV = PV[V,max], CV[min,max] only Eng/Dutch
  // PV: [V,max]-[0,N] and CV [min,max]-[0,N]

  // See Appendix H for the methods used to make decisions
  // using fuzzy sets and chromosomes
  // Decision are what to set starting price (SP) and reserve price (RP)

  // methods for making decisions ... rest of code ...


  // ***********************************************
  // Read Tagged Data from agent script

  public String Rdata(StringBuffer Buffer, String tag, int count){

    // these are used to extract markup data from the script
    String data = null;
    String Script = Buffer.toString();
    int tag_Length = tag.length()+2;
    int Begin = Script.toString().indexOf("<"+tag+">");
    int End = Script.toString().indexOf("</"+tag+">");

    for(int i=1;i<count;i++){
      Begin = Script.toString().indexOf("<"+tag+">",End);
      End = Script.toString().indexOf("</"+tag+">",Begin);
    }
```

```
  try{
    data = Script.toString().substring(Begin+tag_Length,End);
  }
  catch(StringIndexOutOfBoundsException e){;}

  return data;
}

// ***********************************************
// Write Tagged Data i.e. modify agent script

public StringBuffer Wdata(StringBuffer Buffer, String tag, String data){

  // used to update markup data in the script
  // have to break string into three; remove old data
  // first + new data + third
  String oldscript = Buffer.toString();
  StringBuffer newscript = new StringBuffer("");
  int tag_Length = tag.length()+2;
  int Begin = Script.toString().indexOf("<"+tag+">");
  int End = Script.toString().indexOf("</"+tag+">");

  try{
    newscript = new StringBuffer(oldscript.toString().substring(0,Begin+tag_Length)
                                 + data + oldscript.toString().substring(End));
  }
  catch(StringIndexOutOfBoundsException e){;}

  return newscript;
}

// ***********************************************
// CONSTRUCTOR

public Seller(Manager i, String Seller_ID, String data){

  // New Thread Created
  super(Seller_ID);
  this.thread_ID = new String(Seller_ID);
  this.Script = new StringBuffer(data);
  this.ias = i;
}

public void run(){

  ias.IAS_Status.setText("Checking Seller Script");
  // Main Markup Elements are:
  // <ID><AUCTION ASSUMPTIONS><GENOTYPE><FUZZY PARTITION><MEMORY>
  // read in mark up data
  String Status = Rdata(Script,"STATE",1);
  String creator = Rdata(Script,"CREATOR",1);
  String address = Rdata(Script,"ADDRESS",1);
  String DateTime = Rdata(Script,"DATETIME",1);

  // rest of code for reading in data ...

  // add Seller agent to Manager's auctioneer monitor vector
  // Seller's auction details are held by auctioneer monitor
  // need to pick out correct element for this seller to read/write
  // sindex is this seller's reference to its Managers auctioneer monitor
  // Monitors variables read from file
  int approxPrice = (minVal+maxVal)/2-(minSD+maxSD)/2;
  Monitor ref = new Monitor( "REGISTERING",creator,Rdata(Script,"AUCTION METHOD",1),
                             address,Rdata(Script,"START",1),
                             Seller_ID,Rdata(Script,"LOT ID",1),
                             creator,address,approxPrice);

  ias.monitors.addElement(ref); // update Manager's auctioneer monitors vector
  int sindex = ias.monitors.size()-1;

  // Seller's states, transitions, events and actions

  for(;;){
    ref = (Monitor) ias.monitors.elementAt(sindex);
    Status = Rdata(Script,"STATE",1);
```

```
if (Status.equals("WAITING")){

  int ncount = 0;

  for(;;){
    now = new Date();
    if (now.after(start)){
      this.Script = Wdata(Script,"STATE","PREPARING");
      break;
    }
    else{
      // accept new registering bidder
      // bidders update the correct Monitor for this seller ID
      // seller reads its monitor & adds mark up to its script
      ref = (Monitor) ias.monitors.elementAt(sindex);
    }
  }
}
else if (Status.equals("PREPARING")){

  this.Script = Wdata(Script,"BIDDER COUNT",Integer.toString(ref.bidderCount));

  if ((bc=ref.ReadBidderCount()) > 0){

    // Actions
    // Calculate SP
    // Update own script

    // Beliefs i.e prior to Bidding
    int beliefRange = maxVal-minVal;
    int beliefSD = (maxSD+minSD)/2;      // average
    int beliefVal = (maxVal+minVal)/2;   // average

     if (aucType.equals("CV")){

       // decide reserve price code ...

       if (aucMethod.equals("ENGLISH")){
          // decide starting price code ...
       }
       else if (aucMethod.equals("DUTCH")){
          // decide starting price code ...
       }
     }
     else{
       // similarly for PV auctions ...
     }

     // update script and Monitor to set SP tags
     this.Script = Wdata(Script,"STARTING PRICE",Integer.toString(SP));
     this.Script = Wdata(Script,"RESERVE PRICE",Integer.toString(RP));
     ref.SetRP(RP);
     ref.Initialise(SP);

     // Update Seller List
     this.Script = Wdata(Script,"STATE","AUCTIONING");
   }
   else{
      // no bidders present
      this.Script = Wdata(Script,"STATE","COMPLETING");
   }
 }
 else if (Status.equals("AUCTIONING")){

   // Auction item or lot: Four Auctions Methods
   // FPSB/ Vickrey - only one round
   // English Auction ends when all but one say Y
   // Dutch Auction ends when first says Y
   // bidder count should be fixed by now i.e no more can join
   String bidanswers[][] = new String[ref.bidderCount][2];

   if (aucMethod.equals("FPSB")){

     String who;
     int[] bids = new int[bc];
     bidanswers = ref.ReadBidAnswers();
```

```
  for(int i=0;i<bc;i++){
    int bid = Integer.parseInt(bidanswers[i][1]);
    bids[i] = bid;
    who = bidanswers[i][0];
    this.Script = Wdata(Script,"BIDDER ID & BID",who + ": " + bid);
  }

  a = HIGH(bids)[1];
  a2  = HIGH2(bids)[1];
  b = LOW(bids);
  int w = HIGH(bids)[0];

  outcome = bidanswers[w][0] + " Bid = " + Integer.toString(a);
  this.Script = Wdata(Script,"HIGH BID",Integer.toString(a));          // highest
  this.Script = Wdata(Script,"SECOND HIGH BID",Integer.toString(a2)); // 2nd high
  this.Script = Wdata(Script,"LOW BID",Integer.toString(b));           // lowest
}
else if (aucMethod.equals("VICKREY")){

  String who;
  int[] bids = new int[bc];
  bidanswers = ref.ReadBidAnswers();

  for(int i=0;i<bc;i++){
    int bid = Integer.parseInt(bidanswers[i][1]);
    bids[i] = bid;
    who = bidanswers[i][0];
    this.Script = Wdata(Script,"BIDDER ID & BID", who + ": " + bid);
  }

  a = HIGH(bids)[1];
  a2  = HIGH2(bids)[1];
  b = LOW(bids);
  int w = HIGH2(bids)[0];

  outcome = bidanswers[w][0] + " Bid = " + Integer.toString(a);
  this.Script = Wdata(Script,"HIGH BID", Integer.toString(a));         // highest
  this.Script = Wdata(Script,"SECOND HIGH BID",Integer.toString(a2)); // 2nd high
  this.Script = Wdata(Script,"LOW BID",Integer.toString(b));           // lowest
}
else if (aucMethod.equals("ENGLISH")){

  a2 = SP;          // asking Bid
  a = 0;            // current accepted
  int i;
  int stillin = 0;
  int top=-1;
  String ID;
  String answer = "N"; // default

  for(;;){
    // note for bidders range = current high - SP
    this.Script = Wdata(Script,"HIGH BID",Integer.toString(a2));
    ref.SetAskingBid(a2);
    bidanswers = ref.ReadBidAnswers();

    for(i=1;i<=bc;i++){
      answer = bidanswers[bc-i][1];
      this.Script = Wdata(Script,"BIDDER/BID",bidanswers[bc-i][0]+":"+answer);

      if (answer.equals("Y")){
        top = bc-i;
        stillin++;
        ID = bidanswers[bc-i][0];
        a=a2;
      }
    }

    if (stillin < 1 || (stillin==1 && a2>RP)){

      ref.SetAskingBid(-1);
      if (top==-1){
        outcome = "NO BID OFFERS";
      }
      else{
        outcome = bidanswers[top][0] + " Bid = " + Integer.toString(a);
        Wdata(Script,"TOP BIDDER",bidanswers[top][0]);
```

```
            this.Script = Wdata(Script,"HIGH BID",Integer.toString(a2));
          }
          break;
        }
        stillin=0;
        a2+=deltaBid;
      }

      this.Script = Wdata(Script,"HIGH BID",Integer.toString(a2));
    }
    else if (aucMethod.equals("DUTCH")){

      String answer = "N";       // default;
      a = SP;                    // asking bid
      boolean taker = false;

      for(;;){
        ref.SetAskingBid(a);
        this.Script = Wdata(Script,"HIGH BID",Integer.toString(a));
        bidanswers = ref.ReadBidAnswers();

        for(int i=1;i<=bc;i++){
          answer = bidanswers[bc-i][1];
          this.Script = Wdata(Script,"BIDDER/BID",bidanswers[bc-i][0]+":"+answer);

          if (answer.equals("Y") || a<RP){
            ref.SetAskingBid(-1);
            if (a<RP) outcome = "NOTAKERS";
            else outcome = bidanswers[bc-i][0] + " Bid = " + Integer.toString(a);
            Wdata(Script,"TOP BIDDER" + bidanswers[i][0]);
            taker = true;
            break;
          }
        }

        if (taker) break;
        else a-=deltaBid;
      }
    }
    // when auction is over
    this.Script = Wdata(Script,"STATE","COMPLETING");
  }
  else if (Status.equals("COMPLETING")){

    String which = "";
    if (bc == 0 ){
      // Update Manager Seller Window with No Bidders - Nosale
      outcome = "NO BIDDERS";
    }
    else{
      if (a >= RP) which = "SALE TO: "; else which = "NOSALE : ";
    }

    // write outcomes: bidder ID and how much was bid
    ref.SetOutcome(which + outcome);
    // if bids are equal only first bid counts
    this.Script = Wdata(Script,"AUCTION OUTCOME",which + outcome);

    outcome = Rdata(Script,"AUCTION OUTCOME",1);
  }
  else{
    // Seller error - State not recognised
    // Display Window with Seller Malfunction!
    ias.IAS_Status.setText(Seller_ID + "  Malfuction! -
    Abort Seller and Auction!");
    this.stop();
  }
 }
}
}
```

```
// ****************************
// BIDDER AGENT INTERPETER CLASS
// ****************************


class Bidder extends Thread{

  StringBuffer Script;
  String thread_ID, Bidder_From;
  Manager ias;

  String CGI_Status, Status, creator, address, DateTime,
         Bidder_ID, LOTID, aucType, aucMethod,s1,s2,s3;

  int    minVal, maxVal, minSD, maxSD,p1,p2,p3,fvLen,
         GAValMin, GAValMax, GARangeMin,GARangeMax,GASDMin,GASDMax;

  // Similar methods used by Bidder agent to decide Bid and Scale-down
  // See Appendix H for simulated agent

  // CONSTRUCTOR
  public Bidder(Manager i, String Bidder_ID, String data, String source){

    super(Bidder_ID);
    this.thread_ID = new String(Bidder_ID);
    this.Script = new StringBuffer(data);
    this.ias = i;
    this.Bidder_From = new String(source);
  }

  public void run(){

    ias.IAS_Status.setText("Checking Bidder Script");
    CGI_Status = "NONE";
    Status = Rdata(Script,"STATE",1);
    creator = Rdata(Script,"CREATOR",1);
    address = Rdata(Script,"ADDRESS",1);
    DateTime = Rdata(Script,"DATETIME",1);
    Bidder_ID = creator + " " + address + " " + DateTime;

    // Extract Auction Assumptions from script
    LOTID = Rdata(Script,"LOT ID",1);
    aucType = Rdata(Script,"AUCTION TYPE",1);
    aucMethod = Rdata(Script,"AUCTION METHOD",1);

    // rest of code for reading in data ...

    // when bidder visits its own addresses are added locally
    boolean newAdd = false;

    for (int i=0;i<(refs+1);i++){

      String a,b,c;
      newAdd = true;
      if (i==0){
        a = creator;
        b = address;
      }
      else{
        a = Rdata(Script,"USER",i);
        b = Rdata(Script,"IA ADDRESS",i);
      }

      for (int j=0;j<ias.auctionAddresses.size();j++){

        c = (String) ias.auctionAddresses.elementAt(j);
        if (c.equals(a+":::"+b)){
          newAdd = false;
          break;
        }
      }

      if (newAdd) ias.auctionAddresses.addElement(a+":::"+b);
    }
```

```
Date now = new Date();
Monitor R;

boolean newAddress = false;
boolean existingAddress = false;

// Bidder's states, transitions, events and actions

for(;;){

  now = new Date();
  Status = Rdata(Script,"STATE",1);

  if (Status.equals("STARTING")){

    refs = Integer.parseInt(Rdata(Script,"IA REFS COUNT",1));
    Schedule[0][1] = Rdata(Script,"CREATOR",1);
    Schedule[0][2] = Rdata(Script,"ADDRESS",1);

    for(int i=1;i<(refs+1);i++){
      Schedule[i][0] = Rdata(Script,"IA REF STATUS",i);
      Schedule[i][1] = Rdata(Script,"USER",i);
      Schedule[i][2] = Rdata(Script,"IA ADDRESS",i);
      Schedule[i][3] = Rdata(Script,"START",i);
      Schedule[i][4] = Rdata(Script,"APPROX PRICE",i);

      System.out.println(Schedule[i][1]+"%%%"+Schedule[i][2]);
    }

    // IA REF TODO status
    for (int i=1;i<(refs+1);i++){

      existingAddress=false;

      if (Schedule[i][0].equals("TODO")){

        toAddress = Rdata(Script,"IA ADDRESS",i);
        toUser = Rdata(Script,"USER",i);
        if (toUser.equals(creator) && toAddress.equals(address)) break;
        marker = i;
        Script = Wdata(Script,"IA REF STATUS","NEXT",marker);
        existingAddress = true;
        break;
      }
    }

    if (searchUntil.after(now) && existingAddress){

      // then attempt to move - Checking Destination is OK
      Date Timer; // set timer
      // need a seperate thread for HTTP call.
      // Interpreter will now time the User
      // for Bidder Acceptance or Reject Response

      ThreadGroup posters = new ThreadGroup("Excluded Threads");
      Poster poster = new Poster(poster,this.ias,creator,address,toUser,toAddress);
      poster.setPriority(Thread.MIN_PRIORITY);
      poster.start();

      for(;;){

        Timer = new Date();
        try{
          Thread.currentThread().sleep(1000);
        }
        catch(InterruptedException e){;}
        int waiting_period = 10;
        if (Math.abs(Timer.getMinutes()-now.getMinutes()) > waiting_period) break;
        if (poster.posterFinished){
          CGI_Status = poster.reply;
          // CGI_Status set to poster.reply
          break;
        }
      }

      poster.stop(); // poster killed off

      if (poster.posterFinished && CGI_Status.endsWith("ACCEPTED")){
```

```
      Script = Wdata(Script,"STATE","MOVING",1);
    }
    else if (poster.posterFinished && CGI_Status.endsWith("REJECTED")){
      Script = Wdata(Script,"IA REF STATUS","REJECTED",marker);
    }
    else{
      Script = Wdata(Script,"IA REF STATUS","ABANDON",marker);
    }
  }
  else if (!existingAddress && searchUntil.after(now)){
    Script = Wdata(Script,"STATE","COOPERATING",1);
  }
  else if (searchUntil.before(now)){
    Script = Wdata(Script,"STATE","DECIDING",1);
  }
}
else if (Status.equals("COOPERATING")){

  // pick new auction address and change to STARTING
  marker = Integer.parseInt(Rdata(Script,"IA REFS MARKER",1));
  //initially = refs but ++ when new address - retrace
  int ele = ias.auctionAddresses.size();
  newAddress=false;

  for(int i=0;i<ele;i++){

    String A = (String) ias.auctionAddresses.elementAt(i);
    // check it is a new address
    newAddress = true;

    for (int j=0; j<(refs+1);j++){
      if (A.equals(Schedule[j][1]+":::"+Schedule[j][2])){
          newAddress = false;
          break;
      }
    }

    if (newAddress){
      toUser = A.substring(0,A.indexOf(":::"));
      toAddress = A.substring(A.indexOf(":::")+3);
      String newMarkup = "\n\t<IA REF>\n" + "\t\t<IA REF STATUS>" +"TODO" +
                         "</IA REF STATUS>\n" + "\t\t<USER>" + toUser+"</USER>\n" +
                         "\t\t<IA ADDRESS>" + toAddress + "</IA ADDRESS>\n" +
                         "\t\t<START>" + "</START>\n" + "\t\t<APPROX PRICE>" +
                         "</APPROX PRICE>\n" + "\t</IA REF>";

      Script.insert(Script.toString().
                        lastIndexOf( "</IAREF>") +"</IA REF>". length(),newMarkup);
      refs=refs+1;
      Script = Wdata(Script,"IA REFS COUNT",Integer.toString(refs),1);
      marker = refs;
      Script = Wdata(Script,"IA REFS MARKER",Integer.toString(marker),1);
      break;
    }
  }

  if (newAddress) Script = Wdata(Script,"STATE","STARTING",1);
  else if (!newAddress && marker>1){
    // retrace
    existingAddress = true;
    marker = marker-1;
    Script = Wdata(Script,"IA REFS MARKER",Integer.toString(marker),1);
    toUser = Rdata(Script,"USER",marker);
    toAddress = Rdata(Script,"IA ADDRESS",marker);
    Script = Wdata(Script,"STATE","STARTING",1);
  }

  else Script = Wdata(Script,"STATE","DECIDING",1);
  // i.e no more address links from any auction
}
else if (Status.equals("MOVING")){

  Script = Wdata(Script,"STATE","OBSERVING",1);
  Script = Wdata(Script,"IA REFS MARKER",Integer.toString(marker),1);
  ias.IAS_Status.setText("A Bidder Has Left");
  // Bidder agent is posted and this thread is stopped
  CGI_Status = ias.IAS_Poster( creator,address,"BIDDER",toUser,
                               toAddress,Script.toString(),"","","","");
```

```
  this.stop();
}
else if (Status.equals("OBSERVING")){

  boolean goodAuction = false;
  R = null;

  for (int i=0; i< ias.monitors.size();i++){

    R = (Monitor) ias.monitors.elementAt(i);
    Date d = new Date(R.start);

    if (LOTID.equals(R.lotID) && now.before(d) &&

      aucMethod.equals(R.method) && d.before(startBy)){
      int test=-1;
      goodAuction = true;
      // either immediate bargin or have already decided

      if ( barginPrice>R.approxPrice || ((test=Script.toString()
           indexOf("CHOSEN AUCTION")) !=-1 && ((R.start).
           equals(Rdata(Script,"START",marker))))){

        Script = Wdata(Script,"STATE","REGISTERING",1);
        sindex = i; // seller index
        Seller_ID = ((Monitor) ias.monitors.elementAt(sindex)).sellerID;
        start = R.start;
      }
      else Script = Wdata(Script,"STATE","STARTING",1);

      String iarefstate = "ACCEPTABLE";

      if (test == -1){
        if (barginPrice>R.approxPrice) iarefstate = "CHOSEN AUCTION";

        String newMarkup = "\t<IA REF STATUS>" + arefstate + "</IA REF STATUS>\n" +
                           "<USER>"+R.User+ "</USER>\n"+
                           "\t<IAADDRESS>" +  R.address + "</IA ADDRESS>\n" +
                           "<START>" + R.start + +"</START>\n" +
                           "\t<APPROX PRICE>" + R.approxPrice+ "</APPROX PRICE>";

        Script = Wdata(Script,"IA REF",newMarkup,marker);
      }
      break;
    }
  }

  if (!goodAuction){
    Script = Wdata(Script,"IA REF STATUS","NO SUITABLE AUCTIONS",marker);
    Script = Wdata(Script,"STATE","STARTING",1);
  }
}
else if (Status.equals("DECIDING")){

  // go through IA REFS pick lowest price auctions start before date
  int minPrice = Integer.MAX_VALUE;
  boolean auctionFound=false;

  for(int i=1;i<(refs+1);i++){
    String s = Rdata(Script,"IA REF STATUS",i);
    String m = Rdata(Script,"USER",i);
    String a = Rdata(Script,"IA ADDRESS",i);
    String t = Rdata(Script,"START",i);
    String ps = Rdata(Script,"APPROX PRICE",i);

    if (s.equals("ACCEPTABLE") && (new Date()).before(new Date(t))){

      int price = Integer.parseInt(ps);
      if (price<minPrice){
        minPrice = price;
        toUser = m;
        toAddress = a;
        marker = i;
        Script = Wdata(Script,"IA REFS MARKER",Integer.toString(i),1);
      }
      auctionFound = true;
    }
  }
```

```
  if (auctionFound){
    if (location.equals(toUser+":::"+toAddress)) Script =
         Wdata(Script,"STATE","OBSERVING",1);
    else Script = Wdata(Script,"STATE","MOVING",1);

    Script = Wdata(Script,"IA REF STATUS","CHOSEN AUCTION",marker);
  }
  else{
    Script = Wdata(Script,"STATE","COMPLETING",1);
  }
}
else if (Status.equals("REGISTERING")){

  // Register with selected seller by updating ias.monitors
  bindex = ((Monitor) ias.monitors.elementAt(sindex)).Register(Bidder_ID);

  // update script with Auction Log - not the first time!
  String st = Script.toString();

  if (st.indexOf("AUCTION LOG") == -1 || played>0){

    String newMarkup = "\n\n\t<AUCTION LOG>\n"+
                       "\t<LOCATION>"+location+"</LOCATION>\n"+
                       "\t<OUTCOME></OUTCOME>\n" +
                       "\t<STARTING PRICE></STARTING PRICE>\n"+
                       "\t<RESERVE PRICE></RESERVE PRICE>\n"+
                       "\t<BID OFFER></BID OFFER>\n"+
                       "\t<SCALEDOWN></SCALEDOWN>"+"</AUCTION LOG>";

    Script.insert(Script.toString().lastIndexOf("</AUCTION LOG>")+
                  "</AUCTION LOG>".length(),newMarkup);
  }
  else{
    // i.e only for first auction
    Script = Wdata(Script,"LOCATION",location,1);
  }

  Date d = new Date(start);

  for(;;){
    now = new Date();

    if (now.after(d)){
      Script = Wdata(Script,"STATE","BIDDING",1);
      break;
    }
  }
}
else if (Status.equals("BIDDING")){

  // Beliefs i.e prior to Bidding
  int beliefRange = maxVal-minVal;
  int beliefSD = (maxSD+minSD)/2;
  int beliefVal = (maxVal+minVal)/2;
  int averageVal=0;
  int askingBid=0;
  int bidamount=0;
  int scaledown, range;

  R = (Monitor) ias.monitors.elementAt(sindex);

  if (aucMethod.equals("ENGLISH") || aucMethod.equals("DUTCH")){
    SP = R.ReadAskingBid();
    askingBid = SP;
    Script = Wdata(Script,"STARTING PRICE",Integer.toString(SP),played+1);
  }

  for(;;){

    if (aucType.equals("PV")){

      CBID = Rdata(Script,"CHROMO BID",1);
      for (int i=0; i<fvLen;i++) PBID[i]=Integer.parseInt(CBID.substring(i,i+1));

      if (aucMethod.equals("FPSB") || aucMethod.equals("VICKREY"){
        // decide bid code ..
```

```
      bidAnswer = Integer.toString(bidamount);
      R.SetBidAnswer(bidAnswer,bindex);
      Script = Wdata(Script,"BID OFFER",bidAnswer,played+1);
      break;
    }
    else if (aucMethod.equals("ENGLISH") || aucMethod.equals("DUTCH")){

      // update beliefs
      if (aucMethod.equals("ENGLISH") && askingBid<maxVal){
        averageVal=(askingBid+maxVal)/2;
        range = maxVal-askingBid;
      }
      else if (aucMethod.equals("DUTCH") && askingBid>minVal){}
      else{
        averageVal = askingBid;
        range = 0;
      }

      // decide bid code ...
      if (aucMethod.equals("ENGLISH"))
        if (bidamount > askingBid) bidAnswer = "Y";
        else bidAnswer = "N";
      else if (aucMethod.equals("DUTCH"))
        if (askingBid > bidamount) bidAnswer = "N";
         else bidAnswer = "Y";

      R.SetBidAnswer(bidAnswer,bindex);
      askingBid = R.ReadAskingBid();
      if (askingBid == -1) break;
    }
  }
  else{
    // CV auctions
    CBID = Rdata(Script,"CHROMO BID",1);
    CSD = Rdata(Script,"CHROMO SCALEDOWN",1);

    for (int i=0; i<fvLen; i++){
      PBID[i] = Integer.parseInt(CBID.substring(i,i+1));
      PSD[i] = Integer.parseInt(CSD.substring(i,i+1));
    }

    if (aucMethod.equals("FPSB") || aucMethod.equals("VICKREY")){
      // decide bid and scale-down code ...
      bidAnswer = Integer.toString(bidamount);
      R.SetBidAnswer(bidAnswer,bindex);
      Script = Wdata(Script,"BID OFFER",bidAnswer,played+1);
      break;
    }
    else if (aucMethod.equals("ENGLISH") || aucMethod.equals("DUTCH")){

      // update beliefs
      if (aucMethod.equals("ENGLISH") && askingBid<maxVal){
        averageVal= (askingBid+maxVal)/2;
        range = maxVal-askingBid;
      }
      else if (aucMethod.equals("DUTCH") && askingBid>minVal){}
      else{
        averageVal = askingBid;
        range = 0;
      }

      // decide bid and scaledown code ...

      if (aucMethod.equals("ENGLISH"))
        if (bidamount > askingBid) bidAnswer = "Y";
        else bidAnswer = "N";
      else if (aucMethod.equals("DUTCH"))

      if (askingBid > bidamount) bidAnswer = "N";
      else bidAnswer = "Y";

      R.SetBidAnswer(bidAnswer,bindex);
      askingBid = R.ReadAskingBid();

      if (askingBid == -1) break;
    }
  }
```

```
        Script = Wdata(Script,"BID OFFER",
        Integer.toString(bidamount),played+1);
      }

      String result = R.ReadOutcome();
      Script =Wdata(Script,"RESERVE PRICE",Integer.toString(R.reservePrice),played+1);
      Script = Wdata(Script,"AUCTIONS PLAYED",Integer.toString(played+1),1);
      Script = Wdata(Script,"OUTCOME",result,played+1);

      // if not winning bidder then go to STARTING state
      if (result.startsWith("NOSALE")){
        Script = Wdata(Script,"STATE","DECIDING",1);
        Script = Wdata(Script,"IA REF STATUS","NOSALE AUCTION",marker);
      }
      else if (!result.substring(9).startsWith(this.thread_ID)){
        Script = Wdata(Script,"STATE","DECIDING",1);
        Script = Wdata(Script,"IA REF STATUS","LOST AT AUCTION",marker);
      }
      else{
        // bidder won then go to COMPLETING state
        Script = Wdata(Script,"STATE","COMPLETING",1);
        Script = Wdata(Script,"IA REF STATUS","WON AT AUCTION",marker);
      }
    }
    else if (Status.equals("COMPLETING")){

      Script = Wdata(Script,"STATE","HOME",1);

      CGI_Status = ias.IAS_Poster(creator,address,"BIDDER"creator,
                                  address,Script.toString(),"","","","","");

      this.stop();
    }
    else if (Status.equals("HOME")){

      outcome = Rdata(Script,"OUTCOME",1);
      ias.IAS_Status.setText("A BIDDER HAS FINISHED");
      this.stop();
    }
    else {
      // Problem
      ias.IAS_Status.setText(Bidder_ID + "Malfuction! - Bidder Aborted");
      this.stop();
    }
  }
 }
}
```

Appendix H – Partial Source Code for the Auction Simulator

This appendix shows the main structure and methods for simulated Bidder agents and the FGA-based auction simulator. Since simulated Seller agents use identical methods to simulated Bidder agent the code for simulated Seller agents is not shown. However, since the auction simulator is a long program only the main structure is shown with its main methods.

```
// ****************************
// SIMULATED BIDDER AGENT CLASS
// ****************************

class SimBidder{

  Random r;

  // fuzzy input partitioning
  int p1,p2,p3;
  int len;
  // gene's alleles for partitioning output decisions
  int e1,e2;
  // decision chromosomes
  int[] BID;
  int[] SD;

  // fitness
  float fitness;
  int played, cash, pl;
  // played, won, nosale
  int auction[] = {0,0,0};
  int bid;

  public int DecFV2(int[] chromo, int highexpr, float[] fv1,
                    float[] fv2, int a, int b,int c,int d){

    int l1 = fv1.length;
    int l2 = fv2.length;
    int N = highexpr-1; // low gene expr is always 0

    float threshold = 0;
    int loci=0;
    int locj=0;
    float dom = 0;

    for(int i=0; i<l1;i++)
      for(int j=0;j<l2;j++){
        float v = (float) fv1[i]*fv2[j];
          if (dom < v){
            dom = v;
            loci = i;
            locj = j;
          }
          threshold+=((float)chromo[i*l2+j]*v);
      }

    // Decide on RP or SP
    // [a,b]-[0,N-1]
    // round down
    int minbid = a + loci*(b-a)/l1;
    int maxbid = a +(loci+1)*(b-a)/l1;
    int maxrange = c + (locj+1)*(d-c)/l2;
    minbid = minbid - maxrange/2;
    maxbid = maxbid + maxrange/2;

    return (int)(minbid + (float)(threshold/N)*(maxbid-minbid));
  }
```

```
// ******************************************************

public int DecFV3(int[] chromo, int highexpr, float[] fv1, float[] fv2,
                  float[] fv3, int a, int b,int c, int d, int e, int f){

  int l1 = fv1.length;
  int l2 = fv2.length;
  int l3 = fv3.length;
  int N = highexpr-1; // low gene expr is always 0

  float threshold = 0;
  int loci=0;
  int locj=0;
  int lock=0;
  float dom = 0;

  for(int i=0; i<l1;i++)
    for(int j=0;j<l2;j++)
      for(int k=0;k<l3;k++){
        float v = (float) fv1[i]*fv2[j]*fv3[k];
          if (dom < v){
            dom = v;
            loci = i;
            locj = j;
            lock = k;
          }

          threshold+=((float)chromo[i*l2*l3+j*l3+k]*v);
      }

  // Decide on RP or SP
  // [a,b]-[0,N-1]

  int minbid = a + loci*(b-a)/l1;
  int maxbid = a +(loci+1)*(b-a)/l1;
  int minsd =  e + lock*(f-e)/l3;
  int maxsd =  e +(lock+1)*(f-e)/l3;
  int maxrange = c + (locj+1)*(d-c)/l2;
  minbid = minbid - maxrange/2 - maxsd;
  maxbid = maxbid + maxrange/2 - minsd;

  return (int)(minbid + (float)(threshold/N)*(maxbid-minbid));
}


public int DecFV3SD(int[] chromo, int highexpr, float[] fv1,
                    float[] fv2, float[] fv3, int e, int f){

  int l1 = fv1.length;
  int l2 = fv2.length;
  int l3 = fv3.length;
  int N = highexpr-1; // low gene expr is always 0

  float threshold = 0;
  int loci=0;
  int locj=0;
  int lock=0;
  float dom = 0;

  for(int i=0; i<l1;i++)
    for(int j=0;j<l2;j++)
      for(int k=0;k<l3;k++){
        float v = (float) fv1[i]*fv2[j]*fv3[k];
          if (dom < v){
            dom = v;
            loci = i;
            locj = j;
            lock = k;
          }
         threshold+=((float)chromo[i*l2*l3+j*l3+k]*v);
       }

  // Decide on RP or SP
  // [a,b]-[0,N-1]

  int minsd =  e + lock*(f-e)/l3;
  int maxsd =  e +(lock+1)*(f-e)/l3;
  minsd = minsd/2;
```

```
  maxsd = 3*maxsd/2;

  return (int)(minsd + (float)(threshold/N)*(maxsd-minsd));
}

// *******************************************************

public float[] AV(int val, int a,int b){

  // This method converts one agent's environmental factor into a Fuzzy Vector
  // Inputs are:     Val = average bidder valuation; a = min val, b = max val
  // Outputs are:    A string of n float [0,1] values [fuzzy graphs]
  // variable overlap is ignored - just partitioning of p will be used

  float[] fv = new float[p1];
  // initialise

  for (int i=0;i<p1;i++) fv[i] = 0;
  float d = (float) (b-a)/(p1-1);

  if (val < a) fv[0] = 1;

  for(int i=0;i<(p1-1);i++){
      if (val >= (a+i*d) && val < (a+(i+1)*d)){
        fv[i] = (float) -val/d + (float) (1+i+a/d);
        fv[i+1] = (float) val/d - (float) (i+a/d);
      }
  }

  if (val >= b) fv[p1-1] = 1;

  return fv;
}

// *******************************************************

public float[] VR(int var, int a,int b){

  // This method converts one agent's environmental factor into a Fuzzy Vector
  // Inputs are:     Var = bidders valuation range
  // Outputs are:    A string of n float [0,1] values [fuzzy graphs]
  // variable overlap is ignored - just partitioning of p will be used

  float[] fv = new float[p2];
  // initialise
  for (int i=0;i<(p2-1);i++) fv[i] = 0;

  float d = (float)(b-a)/(p2-1);
  if (var < a) fv[0] = 1;

  for(int i=0;i<(p2-1);i++){
    if (var >= (a+i*d) && var < (a+(i+1)*d)){
      fv[i] = (float)-var/d + (float) (1+i+a/d);
      fv[i+1] = (float)var/d - (float) (i+a/d);
    }
  }

  if (var >= b) fv[p2-1] = 1;
  return fv;
}

// *******************************************************

public float[] SDV(int sd, int a,int b){

  // This method converts one agent's environmental factor into a Fuzzy Vector
  // Inputs are:     sd = average bidder scaledown (CV only)
  // Outputs are:    A string of n float [0,1] values [fuzzy graphs]
  // variable overlap is ignored - just partitioning of p will be used

  float[] fv = new float[p3];
  // initialise
  for (int i=0;i<(p3-1);i++) fv[i] = 0;

  float d = (float)(b-a)/(p3-1);
  if (sd < a) fv[0] = 1;
```

```
    for(int i=0;i<(p3-1);i++){
      if (sd >= (a+i*d) && sd < (a+(i+1)*d)){
        fv[i] = (float) -sd/d + (float) (1+i+a/d);
        fv[i+1] = (float) sd/d - (float) (i+a/d);
      }
    }

    if (sd >= b) fv[p3-1] = 1;
    return fv;
  }

  // ******************************************************

  public SimBidder(Simulator_GUI sim, long seed){

    // initially set chromo to random
    r = new Random(seed);
    played = Integer.parseInt(sim.aucs.getText());
    cash = Integer.parseInt(sim.cash.getText());
    fitness =0;
    p1 = 0;

    p1 = Integer.parseInt(sim.part1.getSelectedItem().substring(0,2).trim());
    p2 = Integer.parseInt(sim.part2.getSelectedItem().substring(0,2).trim());
    p3 = Integer.parseInt(sim.part3.getSelectedItem().substring(0,2).trim());
    e1 = Integer.parseInt(sim.bid.getSelectedItem().substring(0,2).trim());
    e2 = Integer.parseInt(sim.scale.getSelectedItem().substring(0,2).trim());

    // if PV auction then no SD partition
    if (sim.type[0].getState()) p3 = 1;
    len = p1*p2*p3;
    int n;

    BID = new int[len];
    SD = new int[len];

    for(int i=0;i<len;i++){
      BID[i] = (int)(e1*r.nextFloat());
      SD[i] =  (int)(e2*r.nextFloat());
    }
  }
}
```

```
// *************************
// FGA-BASED SIMULATOR CLASS
// *************************

import java.awt.*;
import java.net.*;
import java.io.*;
import java.util.*;
import java.lang.*;

// ******************************************************

class Simulator extends Thread{

  Simulator_GUI sim;
  Random r;

  SimSeller[] SellerPop;
  SimBidder[] BidderPop;

  int vgen;    // generations
  int vpopbid; // bidder population
  int vpopsell;// seller population
  int vaucs;   // auction played
  int vopps;   // bidder opponets in auction
  float vxprob;// crossover probability
  float vmprob;// mutation probability
  int vVAL;    // agent's private value (PV auctions)
  int vcash;   // agents initial credit

  // auction environment limits using uniform distributions
  String method,type;
  int minVal, minGAVal;  // randomly setting bidder valuations
  int maxVal, maxGAVal;
  int minRange,maxRange;
  int minGARange,maxGARange;
  int minGASD, maxGASD;
  int av,ra,asd;
  // used to create a real bidder or seller
  String seller;
  String bidder;

  // ******************************************************

  public Simulator(Simulator_GUI s){

    // simple constructor
  }

  // ******************************************************
  // Simulated Bidder agent uses this method to decide its bid

  public int BID(SimBidder sb){

    int bid=0;
    int[] chromo = new int[sb.len];
    chromo = sb.BID;

    if (type.equals("CV")){
       bid = sb.DecFV3(chromo,sb.el,
                       sb.AV(av,minGAVal,maxGAVal),
                       sb.VR(ra,minGARange,maxGARange),
                       sb.SDV(asd,minGASD,maxGASD),minGAVal,maxGAVal,
                              minGARange,maxGARange,minGASD,maxGASD);
       // [a,b] not [a,V] or [MIN,MAX] because V<=MAX
    }
    else{
      bid = sb.DecFV2(chromo,sb.el,
                      sb.AV(av,minGAVal,maxGAVal),
                      sb.VR(ra,minGARange,maxGARange),
                            minGAVal,maxGAVal,minGARange,maxGARange);
    }

    return bid;
  }


  // ******************************************************
```

```
// Simulated bidder agent uses this method to decide its scale-down

public int SD(SimBidder sb){

  int sd =0;
  int[] chromo = new int[sb.len];
  chromo = sb.SD;

  if (type.equals("CV")){
    sd = sb.DecFV3SD(chromo,sb.e2,
                     sb.AV(av,minGAVal,maxGAVal),
                     sb.VR(ra,minGARange,maxGARange),
                     sb.SDV(asd,minGASD,maxGASD),minGASD,maxGASD);
    // not scaledown [minGASD,maxGASD] choose [asd/2,3*asd/2]
  }

  return sd;
}

// *******************************************************
// Simulated Seller agent uses this method to decide its reserve price

public int RP(SimSeller ss){

  int[] chromo = new int[ss.len];
  chromo = ss.RP;
  int rp=0;

  if (type.equals("CV")){
    rp = ss.DecFV3(chromo,ss.e2,ss.AV(av,minGAVal,maxGAVal),
                   ss.VR(ra,minGARange,maxGARange),
                   ss.SDV(asd,minGASD,maxGASD),minGAVal,maxGAVal,
                          minGARange,maxGARange,minGASD,maxGASD);
  }
  else if (type.equals("PV")){
    rp = ss.DecFV2(chromo,ss.e1,ss.AV(av,minGAVal,maxGAVal),
                   ss.VR(ra,minGARange,maxGARange),
                          minGAVal,maxGAVal,minGARange,maxGARange);
  }

  return rp;
}

// *******************************************************
// Simulated Seller agent uses this method to decide its starting price

public int SP(SimSeller ss){

  int sp=0;
  int[] chromo = new int[ss.len];
  chromo = ss.SP;

  if (type.equals("CV")){
    sp = ss.DecFV3(chromo,ss.e1,
                   ss.AV(av,minGAVal,maxGAVal),
                   ss.VR(ra,minGARange,maxGARange),
                   ss.SDV(asd,minGASD,maxGASD),minGAVal,maxGAVal,
                          minGARange,maxGARange,minGASD,maxGASD);
  }
  else{
    sp = ss.DecFV2(chromo,ss.e1,
                   ss.AV(av,minGAVal,maxGAVal),
                   ss.VR(ra,minGARange,maxGARange),
                         minGAVal,maxGAVal,minGARange,maxGARange);
  }
  return sp;
}

// *******************************************************
// This method determines number of children a ranked pair of agent will have

public int Children(int rank, int num){
  float c = (float) (4*num)/(num-2);       // y = m*rank+c
  float m = (float) -8/(num-2);
  return (int)(1 + (float) m*rank + c);
}
```

```
// ********************************************************
// This method crossover the a pair of agents' chromosomes

public int[] CrossChromo(int[] chromo1, int[] chromo2, float prob, int len){

  int[] Child = new int[len];
  int order = (int)(2*r.nextFloat());
  int location=0;

  if ((int)(1000*r.nextFloat()) < (int)(1000*prob)){
    location = (int)(len*r.nextFloat());
    //  gene position i
    if ( order < 1){
      for(int i=0;i<location;i++) Child[i] = chromo1[i];
      for(int i=location;i<len;i++) Child[i] = chromo2[i];
    }
    else{
      for(int i=0;i<location;i++) Child[i] = chromo2[i];
      for(int i=location;i<len;i++) Child[i] = chromo1[i];
    }
  }
  else{
    if(order<1) for(int i=0;i<len;i++) Child[i] = chromo1[i];
    else for(int i=0;i<len;i++) Child[i] = chromo2[i];
  }

  return Child;
}

// ********************************************************
// This method mutates genes within a chromosome

public int[] Mutate(int[] chromo, float prob, int len, int expr){

  // change to integer in [0,n-1]
  // mutation from x to x+1 or x-1
  int n,s;

  for(int gene = 0;gene<len;gene++){
    s = Math.round(r.nextFloat());
    if((int)(10000*r.nextFloat()) < (int)(10000*prob)){
      chromo[gene] = chromo[gene]-1+2*s;
      if (chromo[gene] == -1) chromo[gene] = 1;
      if (chromo[gene] == expr) chromo[gene] = expr-2;
    }
  }

  return chromo;
}

// ********************************************************
// Require a sorting class for an array of simAgents
// rank according to fitness level = -bid + vVAL

public SimSeller[] SortS(SimSeller[] a) throws Exception {

  SimSeller T;

  for (int i = a.length; --i>=0; ){
    boolean swapped = false;

    for (int j = 0; j<i; j++) {
      float f1 = a[j].fitness;
      float f2 = a[j+1].fitness;
      if (f1 < f2) {
          T = a[j];
          a[j] = a[j+1];
          a[j+1] = T;
          swapped = true;
      }
    }
    if (!swapped) break;
  }

  return a;
}
```

```
// ******************************************************
// Similarly for sorting Bidder agents according to their fitness


// ******************************************************
// This method generates a new generation by crossing paired agents' chromosomes

public void NewGenerationB(int pb, float crossprob, float mutprob) throws Exception {

  int childcount = 0;
  int sum = 0;
  SimBidder parent1,parent2;

  // any bidder length will do
  int len = BidderPop[0].len;
  int a = BidderPop[0].e1;
  int b = BidderPop[0].e2;

  // to hold the new chromosomes
  int[][] A = new int[pb][len];
  // if PV auction SD chromosome is redundant
  int[][] B = new int[pb][len];
  SortB(BidderPop);

  for(int rank=0;rank<(pb/2);rank++){
    parent1 = BidderPop[2*rank];
    parent2 = BidderPop[2*rank+1];
    childcount = Children(rank+1,pb);

    for(int child=0;child<childcount;child++){
      if (sum<pb){
        System.arraycopy(CrossChromo(parent1.BID,
                         parent2.BID,crossprob,len),0,A[sum],0,len);
        System.arraycopy(Mutate(A[sum],mutprob,len,a),0,A[sum],0,len);
        System.arraycopy(CrossChromo(parent1.SD,parent2.SD,
                         crossprob,len),0,B[sum],0,len);
        System.arraycopy(Mutate(B[sum],mutprob,len,b),0,B[sum],0,len);
        sum++;
      }
      else break;
    }
  }

  for(int l=0;l<pb;l++){
    System.arraycopy(A[l],0,BidderPop[l].BID,0,len);
    System.arraycopy(B[l],0,BidderPop[l].SD,0,len);
  }

  System.out.println("*** BID Chromosome after Crossover ****");

  for (int j=0;j<pb;j++){
    BidderPop[j].cash = vcash;
    BidderPop[j].pl = 0;
    BidderPop[j].fitness=0;
    for(int u=0;u<3;u++){
      BidderPop[j].auction[u]=0;
    }
  }
}

// ******************************************************
// Similarly for new Seller agent generation
// ******************************************************

// ******************************************************
// main simulator thread

public void run(){

  SellerPop = new SimSeller[vpopsell];
  BidderPop = new SimBidder[vpopbid];

  // set auction method and type
  // evolve sellers or bidders
```

```
    if (agentType.equals("EVOLVING SELLER")){

      for(int g=0;g<vgen;g++){

        // This ranks and pairs, and generates new pop using mut and cross
        if (g!=0) NewGenerationS(vpopsell,vxprob,vmprob);

        // test  each seller in population
        for (int s=0;s<vpopsell;s++){
          // test seller for so many auctions

          for (int a=0;a<vaucs;a++){
            // rest of code for auctions ...
          }
        }
        // output statistics for Seller generation
      }
      // output statistics for all Seller agents
    }
    else if (agentType.equals("EVOLVING BIDDER")){

      for(int g=0;g<vgen;g++){

        // This ranks and pairs, and generates new pop using mut and cross
        if (g>0) NewGenerationB(vpopbid,vxprob,vmprob);

        // test each bidder in population
        for (int b=0;b<vpopbid;b++){
          // each bidder is test for so many auctions

          for(int a=0;a<vaucs;a++){
            // randomly choose bidder opponents bids
            // to randomlily generate U[minVal,maxVal] :
            // maxVal<=maxGAVal and minVal>= minGAVal
            av = minGAVal +(int)((maxGAVal-minGAVal)*r.nextFloat());
            ra = minGARange+(int)((maxGARange-minGARange)*r.nextFloat());
            maxVal = av+ra/2;
            minVal = av-ra/2;

            // randomly choose bids from U[minVal,maxVal]
            // rest of auction code ...
          }
        }
        // output statistics per Bidder generation
      }
      // output statistics for all Bidder agents
    }
  }
}
```

References

[AHO86] Aho, A., R. Sethi and J. Ullman "Compilers Principles, Techniques and Tools", Addison-Wesley, 1986, pp.1-82.

[ALM96] Almaand, G. and V. Jagannathan "Orbix Distributed Object Technology for Java: Management Overview", IONA Technologies Ltd, 1996.

[ARI98] Aridor, Y. and D. Lange "Agent Design Patterns: Elements of Agent Application Design", Proceedings of the Second International Conference on Autonomous Agents, Minneapolis/St. Paul, May 1998.

[AXE81] Axelrod, R. and W. Hamilton "The Evolution of Co-operation", Science, March, 1981.

[BEA96a] Beam, C., A. Segev and J. Shanthikumar "Electronic Negotiation through Internet-based Auctions", CITM Working Paper 96-WP-1019, December 1996.

[BEA96b] Beam, C. and A. Segev "Electronic Catalogues and Negotiations", CITM Working Paper 96-WP-1016, August 1996.

[BEL96] Bell, G., A. Parisi and M. Pesce "The Virtual Reality Modeling Language", Version 1.0 Specification, Aereal Inc., May 1996.

[BER94] Berners-Lee, R., et al "Hypertext Transfer Protocol -- HTTP/1.0", Internet-Draft, IETF, December 1994.

[BES89] Beasley, J. " The Mathematics of Games", Open University Press, 1989, pp.10-43.

[BI96] Bic, L., M. Fukuda and M. Dillencourt "Distributed Computing Using Autonomous Objects", IEEE Computer, August 1996.

[BIC98] Bickmore, T., L. Cook and E. Churchill "Animated Autonomous Personal Representatives", Proceedings of the Second International Conference on Autonomous Agents, Minneapolis/St. Paul, May 1998.

[BIN92] Binmore, K. "Fun and Games: A Text on Game theory", Lexington, Mass.: D.C. Heath, 1992, pp.523-536.

[BRE98] Bredin, J., D. Kotz and D. Rus "Market-based Resource Control for Mobile Agents", Proceedings of the Second International Conference on Autonomous Agents, Minneapolis/St. Paul, May 1998.

[CAS67] Cassady, R. "Auctions and Auctioneering", California University Press, 1967.

[CH92] Chalupsky, H. and T. Finin et al (KQML Advisory Group) "An Overview of KQML: A Knowledge Query and Manipulation Language", Dept. of Computer Science, University of Maryland, Baltimore, MD 21228, April 1992.

[CHA96] Chavez, A. and P. Maes "Kasbah: An Agent Marketplace for Buying and Selling Goods", Proceedings of PAAM'96, 1996, pp.75-90.

[CHU98] Chauhan, D. and A. Baker "JAFMAS: A Multiagent Application Development System", Proceedings of the Second International Conference on Autonomous Agents, Minneapolis/St. Paul, May 1998.

[CHO96] Cheong, Fah-Chun "Internet Agents Spiders, Wanderers, Brokers and Bots", New Riders Publishing, Indianapolis, Indiana, 1996, pp.125-151.

[COH96] Cohen, S. et al "Java Fundamentals", Wrox Press, 1997, pp.161-182.

[COLI97] Collins, J., S. Jamison, M. Gini and B. Mobasher "Temporal Strategies in Multi-Agent Contracting Protocol", Proceedings of the AAAI-97 Workshop on AI in Electronic Commerce, Providence RI, 1997.

[COLI98] Collins, J., B. Youngdahl, S. Jamison, B. Mobascher and M. Gini "A Market Architecture for Multi-agent Contracting", Proceedings of the Second International Conference on Autonomous Agents, Minneapolis/St. Paul, May 1998.

[DAW89] Dawkins, R. "The Selfish Gene", 2nd Edition, Oxford University Press, 1989.

[DEC96] Decker, K, M. Williamson and K. Sycara "Matchmaking and Brokering", The Robotics Institute, Carnegie Mellon University, Pittsburgh, PA 15213.

[DOM93] Domowitz, I. "Automating the Continuous Double Auction in Practice: Automated Trade Execution Systems in Financial Markets in the Double Auction Market", D. Friedman and J. Rust, Editors, SFI Studies in the Sciences of Complexity, Proc. Vol. XIV, Addison-Wesley, 1993, pp.27-60.

[DOW96] Dowd, K. "Getting Connected, the Internet at 56K and Up", O'Reilly and Associates, 1996, pp.175-224.

[DUR94] Durkin, J. " Expert systems: Design & Development", Macmillon, 1994, pp.215-250.

[DW95] Dworman, G., S. Kimbrough and J. Laing "Bargaining in a Three-Agent Coalition Game: An Application of Genetic Programming", The Wharton School, University of Pennsylvania, Philadelphia.

[DW96a] Dworman, G., S. Kimbrough, J. Laing "Implementation of a Genetic Programming System in a Game-theoretic Contexts", The Wharton School, University of Pennsylvania, Philadelphia.

[DW96b] Dworman, G., S. Kimbrough and J. Laing "On Automated Discovery of Models Using Genetic Programming in Game-Theoretic Contexts", Proceedings of the 28th Hawaii International Conference on System Sciences, Volume III: Information Systems: Decision Support and Knowledge-based Systems, J.F. Nunamaker, Jr and R. Sprague, Jr., editors, (IEEE Press, Los Alamitos, CA), pp. 428-438, 1995.

[EPS77] Epstein, R. "The Theory of Gambling and Statistical Logic", Academic Press, 1997, pp.215-402.

[ERI97] Eriksson J., N. Finne and S. Janson "Information and Interaction in MarketSpace - towards an open agent-based market infrastructure", Swedish Institute of Computer Science, SICS, Box 1263, S-16428 Kista, Sweden, 1997.

[FIS93] Fisher, M. and M. Wooldridge "Specifying and Verifying Distributed Intelligent Systems", M. Filgueiras and L. Dumas, editors, Proceedings of the sixth Portuguese Conference on AI, Springer-Verlag, October 1993.

[FLC] Flach, Peter "Logical Approaches to Machine Learning- an Overview", Think Quarterly, Institute of Language Technology and Artificial Intelligence.

[FLA96] Flanagan, David "Java in a Nutshell", O'Reilly and Associates, Febuary 1996, pp.161-171.

[FOR84] Forsyth, R. "Expert Systems: Principles and Case Studies", Chapman & Hall, 1984.

[FRA] Franklin, S. and A. Graesser "Is it an Agent, or just a Program?: A taxonomy for Autonomous Agents" In J. P. Miller, M. Wooldridge and N. R. Jennings, editors, Intelligent Agents III, Springer-Verlag, Germany, 1997, pp. 21-36.

[FRI93] Friedman, D. "The Double Auction Market Institution in Financial Markets in the Double Auction Market", D. Friedman and J. Rust, Editors, SFI Studies in the Sciences of Complexity, Proc. Vol. XIV, Addison-Wesley, 1993, pp.3-25.

[GAL97] Gagliano R. and M. Fraser "Software Agents and the Role of Market Protocols", Dept. of Mathematics and Computer Science, Georgia State University, Atlanta, GA 30303-3083, 1996.

[GAR94] Garfinkel, S. and G. Spafford "Practical UNIX Security", O'Reilly & Associates, June 1994, pp.221-253.

[GIL] Gilbert et al "Intelligent Agent Strategy", IBM Corporation, Research Triangle Park, NC, USA, 1995.

[GOL89] Goldberg, D. "Genetic Algorithms in Search, Optimization and Machine Learning", Addison Wesley, 1989, pp.57-87.

[GON95] Goonatilake, S. and P. Treleaven (Eds.) "Intelligent Systems for Finance and Business", John Wiley and Sons, 1995.

[GOR97] Gorda, B. and G. Wilson "Building and running Online Auctions: Webalog, a tool for enhancing online commerce", Dr Dobbs Journal, October 1997.

[GOS95] Gosling, J. and H. McGilton "The Java Language Environment: A White Paper", Technical Report, Sun Microsystems, 1995.

[GRE89] Garey, M. and W. Freeman "Computers and Intractability: A Guide to the Theory of NP-Completeness", W. H. Freeman and Company, 1989.

[GUN96] Gundavaram, S. "CGI Programming on the World Wide Web", O'Reilly and Associates, March 1996, pp.1-50.

[HAE87] David Harel "Algorithmics: The Spirit of Computing", Addison-Wesley Press, 1987, pp.151-209.

[HAR95] Harrison, C., D. Chess and A. Kershenbaum "Mobile Agents: Are they a good idea?", IBM Research Report, IBM Research Division, March 1995.

[HAS89] Harrison, G. W. "Theory and Misbehavior in First-Price Auctions", American Economic Revue 49 (4) (1989).

[HOL92] Holland, J. "Genetic Algorithms", Scientific America, July 1992.

[HU98] Hu, J. and M. Wellman "Learning about other agents in dynamic multiagent systems", Proceedings of the Second International Conference on Autonomous Agents, Minneapolis/St. Paul, May 1998.

[HUB93] Huberman, B. and N. Glance "Evolutionary Games and Computer Simulations", Proc. National Academy of Science, Vol. 90, August 1993, pp. 7716-7718.

[HUR97] Hurst L, P. Cunningham and F. Somers "Mobile Agents – Smart messages", Proceedings of the 1st International Workshop on Mobile agents, MA97.

[HYT] Introduction to HyTime, Communications of the ACM, November 1991.

[JEN98a] Jennings, N. R., K. P. Sycara and M. Wooldridge "A Roadmap of Agent Research & Development", Journal of Autonomous Agents and Multi-Agent Systems, July 1998.

[JEN98b] Jennings, N. R and M. Wooldridge, editors "Agent Technology: Foundation, Applications and Markets", Springer-Verlag, March 1998.

[KAL96] Kalakota, Ravi and Andrew Whinston "Electronic Commerce - A Manager's Guide", Addison Wesley, 1996, pp.63-121.

[KAR91] Karr, C. "Applying Genetics to Fuzzy Logic", AI Expert, March 1991.

[KEN98] Kendall, E., M. Krishna, C. Pathak and C. Suresh "Patterns in Intelligent and Mobile Agents", Proceedings of the Second International Conference on Autonomous Agents, Minneapolis/St. Paul, May 1998.

[KIM] Kimburgh, Steven "On Automated Discovery of Models using Genetic Programming in Game-theorectic Contexts", The Wharton School, University of Pennsylvania.

[KOK94] Kosko, B. "Fuzzy Thinking: The New Science of Fuzzy Logic", Harper-Collins, 1994, pp.121-200.

[KOS96] Koster, M. "Robots in the Web: threat or treat?", ConneXions, Vol. 9, No. 4, 1995.

[KRE96] Kreculj, N. "Web Browser is lined up to succeed client", Computing, 5th September 1996.

[LAM96] Lamier, J. "Secret Agent", Computing, 1st August 1996.

[MAC98] Mace, S., U. Flhr and T. Graham "Weaving a Better Web", Byte, March 1998.

[MAR97] Marshall, J. "Economist's Auction Theory goes to Market", San Francisco Chronicle, 21st April 1997.

[MCA87] McAfee, R. and J. McMillan "Auctions and Bidding", Journal of Economic Literature, 25:699-754, June 1987.

[MIL82] Milgrom, Paul & Robert Weber "A Theory of Auctions and competitive Bidding", Econometrica, 50:1089-1122, September 1982.

[NOR97] Noriega, P. "Agent Mediated Auctions: The Fishmarket Metaphor", PhD Thesis, University of Barcelona, December 1997.

[NWA96] Nwana, H. "Software agents: An Overview", Knowledge Engineering Review, Vol. 11(3), 1996.

[OLE97] O'Leary, D. "The Internet, Intranets and the AI Renaissance", Computer, Jan. 1997.

[OLI97b] Oliver, J. "On Automated Negotiation and Electronic Commerce", Proceedings of 19th Hawaii International Conference on System Sciences, IEEE Computer Society Press, 1996.

[ORF96] Orfali, Robert, Dan Harkey and Jeri Edwards "The Essential Client/Server Survival Guide", 2nd Edition, Wiley, 1996, pp.379-556.

[PAT90] Patterson, D. "Introduction to AI and Expert Systems", Prentice Hall, 1990, pp.97-105.

[PET96] Petrie, C. "Agent-Based Engineering, the Web and Intelligence", IEEE Expert, December 1996.

[POP96] Popham, Michael "Introduction to SGML", CSW Informatics Ltd, 1996.

[PRA94] Pratt, Ian "Artificial Intelligence", Macmillan, 1994, pp.1-40.

[RAS96] Rasmusen, Eric "Games and Information", 2nd Edition, Blackwell, 1996, pp.293-305.

[ROB85] Robinson, Marc "Collusion and the choice of Auction", Rand Journal of Economics, 16:141-5, Spring 1985.

[ROD98] Rodriguez-Aguilar, J., F. Martin, P. Noriega, P. Gere and C. Sierra "Competitive Scenarios for Heterogeneous Trading Agents", Proceedings of the Second International Conference on Autonomous Agents, Minneapolis/St. Paul, May 1998.

[RUS92] Rust, J., J. Miller and R. Palmer "Behavior of Trading Automata in a Computerized Double Auction Market in Financial Markets: in the Double Auction Market", D.

Friedman and J. Rust, Editors, SFI Stidies in the Sciences of Complexity, Proc. Vol. XIV, Addison-Wesley, 1993, pp. 155-198.

[SAN97] Sandholm, T. "Limitations of the Vickrey Auction in a Computational Multiagent Systems", Proceedings of the Second International Conference on Multiagent Systems (ICMAS-96), Keihanna Plaza, Kyoto, Japan, December 1996.

[SHA92] Shlaer and Mellor "Object Lifecycles Modeling the World in States", Yourdon Press, 1992, pp.133-144.

[SHU71] Shubik, Martin "The Dollar Auction Game: A Paradox in Non-cooperative Behaviour and Escalation", Journal of Conflict Resolution, 15:109-11, March 1971.

[SIM85] Simons, G. "Introducing AI", NCC Publications, 1984.

[SMI92] Smith, Joan "SGML and Related Standards", Ellis Horwood Ltd., 1992.

[SOM96] Sommerville, Ian "Software Engineering", 5th Edition, Addison-Wesley, 1996, pp.3-43.

[SPA96] Spainhour, S. and V. Quercia "Webmaster in a Nutshell", O'Reilly and Associates, 1996, pp.71-174.

[TRA96] Trammell, K., "Workflow without Fear", Byte, April 1996.

[TRE93] Treese G. W. and A. Wolman " X Through the Firewall, and Other Application Relays", Digital Corporation Cambridge Research Lab, May 1993.

[TUR96] Turner, R., A. Douglas and A. Turner "Readme.1st SGML for writers and editors", Prentice Hall, 1996, pp.51-69.

[VIC61] Vickrey, W. "Counterspeculation, Auctions and Competitive Sealed Tender", Journal of Finance, 16:8-37, March 1961.

[WAL92] Wall, L. and R. Schwartz "Programming Perl", O'Reilly & Associates, March 1992, pp. 337-361.

[WAS96] Washburn, Kevin and Jim Evans "TCP/IP running a successful network", Second Edition, Addison-Wesley, 1996.

[WEL98] Wellman, M. and P. Wurman "Real Time Issues for Internet Auctions", In First IEEE Workshop on Dependable and Real-time E-Commerce Systems (DARE-98), Denver, USA, June 1998.

[WHI95] White, J. "Telescript Technology: An Introduction to the Language", General Magic White Paper GM-M-TSWP3-0495-V1, General Magic, Inc., 420 North Mary Avenue, Sunnyvale, CA 94086, 1995.

[WOD97] Woodcock, J. "Understanding Groupware in the enterprise", Microsoft Press, 1997, pp.1-20.

[WOO91] Wooldridge, M., G. O'Hare and R. Elks "FELINE – A Case Study in the Design and Implementation of a Co-operating Expert System", Proceedings of the International Conference on Expert Systems and their Applications, Avignon, May 1991.

[WOO92] Wooldridge, M. "The Logical Modelling of Computational Mult-Agent Systems", PhD thesis, UMIST, Manchester, October 1992.

[WOO95] Wooldridge, M. and N. Jennings "Agent Theories, Architectures and Languages: A Survey", Intelligent Agents, Wooldridge and Jennings Editors, Berlin: Springer-Verlag 1-22, 1995.

[WOO96] Wooldridge, M. and N. Jennings "Intelligent Agents: Theory and Practice", Knowledge Enginering Review 10(2), 1995.

[WOO98a] Wooldridge, M. and N. Jennings "Pitfalls of Agent-Oriented Development", Proceedings of the Second International Conference on Autonomous Agents, Minneapolis/St. Paul, May 1998.

[WOO98b] Wooldridge, M. "Agents and Software Engineering", In AI*IA Notizie XI(3), September 1998.

[WU98a] Wurman, P., M. Wellman and W. Walsh "The Michigan Internet AuctionBot: A Configurable Auction Server for Human and Software Agents", Proceedings of the Second International Conference on Autonomous Agents, Minneapolis/St. Paul, 1998.

[WU98b] Wurman, P., M. Wellman and W. Walsh "Flexible Double Auctions for Electronic Commerce: Theory and Implementation", Decision Support Systems, July 1998.

[ZAD74] Zadeh A. "Fuzzy Sets: Their Applications to Cognitive & Decision Problems", New York Academic Press, 1974.

Web Pages (available at the time of writing)

[ADE] Adaptation, Dynamic and Economic Organisation in Multi-agent Systems, <http://cce.sscnet.ucla.edu/papers/devany/249Syl.html>.

[AGO96] Agorics, Inc. "Going, going, gone! A survey of auction types", 1996, <http://www.agorics.com/agorics/auctions/new.html>.

[ARZ] Arizona WWWeb Online, <http://www.azww.com/mall>.

[AS] Michigan AuctionBot links, <http://auction.eecs.umich.edu/other/online.html>.

[AUC] Web-based Auction Sites, <http://www.businessweek.com/1997/32/b3539124.htm>.

[BEN97] Bernet, B. "Software Agents – A Review", <http://www.doc.mmu.ac.uk/STAFF/B.Berney/research/ag-rev.htm>.

[BIN97] Binmore, K. and N. Vulkan "Applying Game Theory to Automated Negotiation", <http://wcw.emory.edu/POINT/9.10.97/9.10.97.8.html>.

[BOS97] Bosak, J. "XML, Java and the future of the Web", <http://sunsite.unc.edu/pub/sun-info/standards/xml/why/xmlapps.html>.

[BT] Mobile Agent Applications, <http://www.labs.bt.com/library/papers>.

[CGI] Common Gateway Interface, <http://hoohoo.ncsa.uiuc.edu/cgi/intro.html>.

[COL] Coleman, D., "Groupware Engineering", <http://www.collaborate.com/publications/publications.html>.

[CRO96] Crowcroft, J. "WWW - Beneath the Surf", <http:// www.citenet.net/main/info/books/bts/node1.html>.

[EBA] E-bay Online Auction, <http://pages.ebay.com/aw/rules.html>.

[FIE96] Fielding et al "Hypertext Transfer Protocol -- HTTP/1.1", HTTP Working Group, June 1996, <http://www.ics.uci.edu/pub/ietf/http/>.

[FIN] Finin, Tim "Agent Programming and Scripting Languages", <http://www.cs.umbc.edu/agents/technology/asl.shtml>.

[FM98] The Fish Market Project, <http://www.iiia.csic.es/Projects/fishmarket/foundations.html>.

[GIL97] Gilbert D. "Intelligent Agents: The right Information at the right Time", IBM Corporation, Research Triangle Park, NC, USA, <http://www.networking.ibm.com/iag/iaghome.html>.

[HOB] Hobby Markets Intelligent Auction agent, <http://www.hobbymarkets.com/>.

[IAL1] Intelligent Agent Links, <http://www.operant.com/agents.html>.

[IAL2] Intelligent Software Agents, <http://www.sics.se/isl/abc/survey.html>.

[INF] Informar Fish Auction Project, <http://nii.nist.gov/g7/10_global_mp/testbeds/infomar.html>.

[KLK] Klik-Klok Online Dutch auction, <http://www.klik-klok.com>.

[LAN96] Lange, D. and D. Chang "IBM Aglets Workbench Programming Mobile Agents in Java", <http://www.trl.ibm.co.jp/aglets/whitepaper.html>.

[LIN96] Lingnau, Drobnik and Domel "An HTTP-based Infrastructure for Mobile Agents", <http://www.tm.informatik.uni-frankfurt.de/ma/www4-paper.html>.

[LIV] Living systems, Agent controlled Online Auction, <http://www.living-systems.com/>.

[MAT98] Matsumura M. "XML Speeds along in standards land", <http://www.javaworld.com/javaworld/jw-02-1998/jw-02-miko.html>.

[MICH] Michigan Internet AuctionBot, <http://auction.eecs.umich.edu/project.html>.

[MKT] Market Based Multi-agent Systems Resource Page, <http://heron.elec.qmw.ac.uk/~mikeg/text.html>.

[OES] OES Corp., <http://www.oes-inc.com/auctclk.htm>.

[OLI97a] Oliver, J. "A Machine Learning Approach to Automated Negotiation and Prospects for Electronic Commerce", <http://opim.wharton.upenn.edu/~oliver27/papers/jmis.ps>.

[ONA] Online Auctions Sites, <http://auction.eecs.umich.edu/other/online.html>.

[ONS] Onsale Web-based auction, <http://www.onsale.com>.

[RES97] Resnick, P. "Roles for Electronic Brokers", MIT, Cambridge, MA, <http://ccc.mit.edu/CCSWP179.html>.

[RO95] Robinson, D. and the Apache Group "Apache: An HTTP Server, Reference Manual", <http://www.apache.org/>.

[TFA] Teleflower auctions, <http://kambil.stern.nyu.edu/teaching/cases/auction/flowers.html>.

[UMB] UMBC AgentWeb, <http://www.cs.umbc.edu/agents/papers/papers.shtml>.

[WWW] World Wide Web Security FAQ, <http://info.webcrawler.com/mak/projects/robots/faq.html>.

[XML] XML, <http://webreview.com/>.