Genghis - A Multiagent Carpooling System

Amit B. Kothari[1]
B.Sc. in Computer Science

May 11, 2004

[1]Particular acknowledgements to Owen Cliffe, my supervisor Dr. Julian Padget, Dr. Alwyn Barry and Walter Barbera-Medina

Genghis - A Multiagent Carpooling System

Submitted by Amit Kothari

COPYRIGHT

Attention is drawn to the fact that copyright of this dissertation rests with its author. The Intellectual Property Rights of the product produced as part of the project belong to the University of Bath (see http://www.bath.ac.uk/ordinances/#intelprop).

This copy of this dissertation has been supplied on condition that anyone who consults it is understood to recognise that its copyright rests with its author and that no quotation from the dissertation and no information derived from it may be published without the prior written consent of the author.

Declaration

This dissertation is submitted to the University of Bath in accordance with the requirements of the degree of Bachelor of Science in the Department of Computer Science. No portion of the work in this dissertation has been submitted in support of an application for any other degree or qualification of this or any other university or institution of learning. Except where specifically acknowledged, it is the work of the author.

This dissertation may be made available for consultation within the University Library and may be photocopied or lent to other libraries for the purposes of consultation.

Abstract

Dynamic carpooling is an important domain which needs an extensible solution for the efficient use of vehicle journeys by commuters as roads become congested. Genghis is the fruit of research into this domain, a Multi Agent System designed through the Gaia methodology and implemented on a FIPA-compliant Jade platform. Two primitive pool types are defined, and we lay the fundamentals of an algorithm for choosing reasonable matches. A model of character in Social ReGreT is used to calculate reputation. The system can incorporate upgrades for journey matching and is a ready foundation for upper layer interfaces like web servlets. It is concluded that the framework of the implemented system can be sufficiently tested from the command line and we create an agent command dictionary.

Contents

1 Introduction **1**

1.1 Aims and Objectives . . . 2

1.2 Pretext . . . 2

2 Literature Survey **3**

2.1 Terms . . . 3

2.2 Introduction to Carpooling . . . 4

2.2.1 Carpool Types . . . 7

2.2.2 Character and Reputation . . . 8

2.3 The Viability of Carpooling . . . 9

2.4 MultiAgent Systems . . . 10

2.5 Development . . . 13

2.5.1 *Jade* . . . 13

2.5.2 Development Methodologies . . . 14

2.5.3 Mapping and graphics . . . 15

3 Carpooling Processes **17**

3.1 Carpooling Ontology . . . 17

3.2 Formulaic Processes . . . 18

3.2.1 Standard . . . 18

3.2.2 Alternatepool . . . 19

4 **MAS Development** **21**
4.1 Agents Design . . . 21
4.2 Temporal Connectives . . . 22
4.3 Basic Requirements . . . 23

5 **Single Journey Matching** **27**
5.1 Overview . . . 27
5.1.1 Contents of Wanted Journey . . . 28
5.2 Decision and Cost Process . . . 28
5.2.1 Level 1 . . . 29
5.2.2 Level 2 . . . 29
5.2.3 Level 3 . . . 29
5.2.4 Level 4 . . . 30
5.3 Return Results . . . 32
5.4 Implementing the DecisionModule Object . . . 32
5.5 Examples . . . 32
5.5.1 Level 3 Example . . . 32
5.5.2 Level 4 Testing . . . 34
5.6 Coding . . . 34

6 **Web interaction and Mapping Overlays** **36**
6.1 Latitude and Longitude . . . 36
6.1.1 Conversion to Distance . . . 38
6.2 ProxyAgents . . . 39

7 **ReGreT and reputation** **42**
7.1 The Problems of Calculating Reputation . . . 42
7.2 Implementing a model for Genghis . . . 43
7.3 Data Privacy Classes for the Schema . . . 45

8 **Testing and Seeding** **47**

8.1 Seed Data . . . 47
8.1.1 TestingAgent . . . 47
8.1.2 Users . . . 47
8.1.3 Journeys . . . 48
8.1.4 Journey-User Associations . . . 48
8.2 Command Dictionary . . . 48
8.3 Simple Testing . . . 50

9 Extensions 51

10 Distributed Genghis 54
10.1 Global Genghis . . . 54
10.2 Versions and Upgrades . . . 56

A MAS and Database Schema 57
A.1 Details . . . 57
A.2 Genghis agents . . . 59
A.3 Database schema . . . 62
A.4 Schema Commentary . . . 63

B Usage Guide 68
B.1 Pre-Requisites . . . 68
B.2 Configuring and Compiling . . . 69
B.3 Running . . . 69

List of Figures

4.1 Basic Contract Net Protocol, implemented in package jade.proto 22
4.2 A human driver or ambivalent adding a new journey 24
4.3 Human requesting a search for a journey - Genghis uses CFP 24
4.4 Human confirming participation in a journey 25
4.5 JourneyNotifyAgent flagging matching journeys to UserAgents 26

5.1 Three parts which add up the distance of a round trip 31
5.2 Three examples scenarios for calculating Level 3 33

10.1 Jade Agent Platform across containers (Reproduced from Jade Programmers Guide) 55

A.1 Schema for Genghis with Privacy codes within an installation 66
A.2 JourneysDB constraints for JourneyID field 67

List of Tables

7.1 Driver Outcomes Database Scheme 45

8.1 Agent command dictionary to be used for launching agents 49

A.1 Abstract and Concrete concepts in the Gaia methodology 57

A.2 Gaia operators for liveness formulas 58

A.3 UserAgent Gaia role model 59

A.4 ProxyAgent Gaia role model 60

A.5 JourneyRoundupAgent Gaia role model 60

A.6 JourneyNotifyAgent Gaia role model 61

Chapter 1

Introduction

This dissertation details the design to various components of a carpooling system, whose primitive parts interact to solve a practical problem.

Chapters 2 - 4 can be characterised as the building blocks for agent communication and protocols, while subsequent Chapters are concerned with dynamic processes.

This document is organised as follows.

Chapter 2 Literature Survey about the domain, MAS and other issues.

Chapter 3 Carpooling as a problem and its component parts and ontology.

Chapter 4 Constructing agents in the MAS using the Gaia methodology. The Contract Net Protocol is used with a discussion about temporal connectives.

Chapter 5 Route matching algorithms to deliver cost.

Chapter 6 Location Mapping, including notes for a web GUI.

Chapter 7 A network with reputations and social interaction.

Chapter 8 Testing the MAS - the performance of single route matching algorithms.

Chapter 9 Extensions for Genghis.

Chapter 10 Practical usage and inter-Genghis connectivity.

This work attempts to address many of the areas where vehicle pooling poses a difficult computational problem, and uses MAS to best effect to separate roles in Genghis. Previous attempts at creating a usable system have been biased to logistical problems [7] or have used simple database-driven websites to give matches using discrete nodes.

Genghis attempts to realise the goal of a *decentralised* and usable system, offering considerable scope for building extensions for optimal route composition and various agent-based notification processes.

1.1 Aims and Objectives

> To study the problem of carpooling, and formalise the essential processes that encompass many kinds of carpooling.
>
> To create a MAS in Jade with supporting components based on a model of the domain understanding.
>
> To identify important algorithms and computational problems we face and make attempts to define or resolve them.

My lack of familiarity with this domain will result in a proportion of resource dedicated to find links in dispersed material, but will encourage a fresh eye for the problem.

1.2 Pretext

Agent interaction diagrams use simple notation to describe process flow between agents, since FIPA AUML Modelling Languages are a work-in-progress.

Items denoted as shown below refer to a *file path* on the **Genghis CD-ROM** or a URL directly associated to this work. In addition, this notation is used for comment *emphasis*.

■ eCopy `/DISSERTATION/PUBLISHING/main.pdf` ■

Agentcities is an appropriate place for developers to host their own web interface to Genghis. This study shows that making a web GUI is straightforward and *crucial* in practice.

■ `http://agentcities.cs.bath.ac.uk/ghengis/` ■

The Genghis Portal, with a mirror of the CD-ROM.

■ `http://www.amitkoth.com/weblog/dissertation/` ■

Chapter 2

Literature Survey

This chapter introduces the domain and surveys material. It starts with a glossary, and discusses carpooling, the viability of carpooling and multiagent systems. I appreciate that there is a wide body of literature on the use of an advanced GIS (Geographic Information System) to facilitate mapping and viewing. The focus of this study is in the multiagent system and not especially the graphical presentation of journeys. A simple co-ordinate grid of latitudes and longitudes would suffice.

2.1 Terms

Car sharing The process of booking a vehicle for personal use. This term will not be used often in this review since it points to a different area of work. Schemes operate widely in Switzerland and Germany [28]. The process is similar to hiring a car but the user is already affiliated to the agency sharing out the car. There is an agent based prototype implementation in JINI [15] called *CarPAcities*. The paper mixes ridesharing and car sharing.

Ridesharing Booking a seat in a person's vehicle for a journey

Carpooler The organisational norm that matches people *looking* for a rideshare and people that *propose* rideshares.

LCPP The Long Term Carpooling Problem. [3]:

> Each user is available to act both as a server and as a client and the objective is to define crews - or user pools - where each user will in turn, on different days pick up the remaining pool members. The

objective here becomes that of maximising pool sizes and minimising total distance travelled by all users when acting as servers

DCPP The Daily Car Pooling Problem. [12]:

> On each day, a number of users (servers) declare their availability for picking up and later bringing back colleagues (clients) on that particular day. The problem is to assign clients to servers and to identify the routes to be driven by the servers in order to service costs and minimise a penalty due to unassigned clients

Organised Hitchhiking Hitchhiking is the informal process of obtaining a rideshare without prior booking. Organised hitchiking is the entire set of formal means by which ridesharing may be achieved. Various *types* of carpooling belong to this set.

Entities Entities are Drivers, Passengers and Ambivalents [33]. An ambivalent is interchangeable between a driver and passenger, and ambivalents are afforded all the characteristics of drivers and passengers.

Carpooling System A methodical solution which allows entities to become active through an interface. The system contains classes of carpool types.

Journey Comprising of information to which an entity associates itself. Journey and trip are analogous. The journey set may consist of an origin, destination, date, time, seats and credit. We have two types of journey. *Regular* journeys have frequency and age parameters which specify that the journeys recurs until a certain time. *Oneoff* journeys occur only once.

Ridesharing groups A set of entities belonging to a rideshare group, which may be private or public. Members of a rideshare group take (exclusively) the same journey type.

2.2 Introduction to Carpooling

Carpooling is a concept which entails a collaborative process to book a seat in a vehicle going to a mutually acceptable destination. In the UK, there is information at different levels of hardness to suggest that the benefits derived from car pooling are attractive, and pooling services make their own claims [25] about the levels of reduction in road congestion. The benefits of pooling people travelling to similar locations are obvious, and include benefits to the environment, urban congestion and cost savings. As part of a regional transport policy [31], it is one way to reduce commuter congestion and could save a great deal of money otherwise allocated for new and widened roads. This vision

is fine, but why hasn't car pooling taken off in a big way? This question of usability is important, addressed separately in Section 2.3.

In the United States it was recognised [30] that carpooling on the web could be effective. The Seattle Smart Traveler [5] was a funded experimental study into the proposition of a pooling scheme at the University of Washington. It followed from earlier work in Seattle on the Bellevue Smart Traveler, concluding in a technical report [8]. A report of the findings of the Seattle Smart Traveler (SST) was conducted by the US Department of Transportation [27] which noted that (Executive Summary pp. VIII) *about 6% actually established a carpool for the requested trip.* The report notes that the majority of users of this system were computer literate but that (p.19) *the system was implemented before the real boom in internet use.* The FTA review is particularly illustrative in that the Appendices include sample web pages, documents and letters. The SST project website [26] contains an offline demonstration and it is clear that there was a lack of thought into user interface design compared to todays norms of standards-compliant markup [27]. Despite these failings, a moderate to low carpool match rate was achieved. This seems to be a recurring theme in many web-based carpooling systems and SST's designers had to accept a certain level of success based on number of users, and Section 2.3 discusses my findings. Strong encouragement of carpooling has been noted in the USA, for example HOV (High Occupancy Vehicle) lanes in Los Angeles [23]. Subsidies, paid parking and vouchers are amongst the other incentives offered by pool scheme hosts.

In the UK most web-based systems only go as far as offering matching routes or nodes for the party looking for a share. An interview with the MD of liftshare.com (one of the UK's largest carpooling services) revealed that they studied a variety of technological ideas involving usage by phone or SMS, but found that real-world use was limited by lack of simplicity and low technological literacy. Public statistics [21] claim that over 34% of journeys registered find a match. It seems that a *successful* system should provide searching and matching like database-driven portals, but can potentially harness the dynamism of a multiagent system through email/SMS. It can be argued (and indeed it is true) that the sheer size of liftshare's operation - which consists of web portals sharing a central dataset, contributes to a higher match rate than SST. A good test of a multiagent system would be at this university, where technological literacy is high. However, we may then expect a journey match rate that is skewed superficially higher due to user familiarity with the GUI. This was the reason why the SST pilot site was the University of Washington, which was also the largest employer at the time in Seattle [26]. A further aspect of workable carpooling that nobody seems to have implemented is that of distributed datasets. Can Genghis be locally implemented and

interconnect with geographically dispersed datasets to achieve user goals? Instead of the notion of a central dataset and thin portals to use it, we later propose a distributed set of Genghis instances that locate each other on a *need to have* basis through a global directory service. This will make the open system widely available to corporate or introvert groups that wish to use it privately.

The carpooling problem may be split academically into two aspects - *LCPP* and *DCPP*. Neither of these are precisely the problem to be solved in this study. We begin by noting that the most basic pool type is the *oneoff* pool where a simple journey is logged, which may recur periodically. The first observation about LCPP is a notion that entities are divided into *pools*, sets grouped by proximity along the route. The objective is to maximise server usage with minimal travel distance. This type of problem occurs when we have an *alternating* pool type where the members partake in journeys to gain a credit for a subsequent journey within the group for themselves. In both cases, we note that potential passengers or ambivalents need to have some 'give' when it comes to timing and location, as an exact match is hard to satisfy. The solution set of LCPP may be put to good use as it represents a case of the alternating carpool, with a static set of members being provided a computed optimal solution which they might choose to follow if they wished. Carbonaro *et al* [3] discuss a heuristic approach to solving LCPP - ANTS (Approximated Non-deterministic Tree Search), an extension to the ANTS system [32]. The paper describes a probabilistic tree search algorithm without a backtracking mechanism. The DCPP is a different problem, the key difference being that the set of clients and servers changes on a daily basis. This is periodically optimal, but requires recomputation every time a new client or server enters the scene. This defeats the purpose of having a dynamic and reasonably *reactive* carpooling system and I don't see practical purpose in computing optimal solutions to oneoff journeys, even if they are return journeys. Through an appropriate multiagent architecture, matches emerge as self evident. Hence we shall not venture into the details of DCPP. It may be sufficient to regenerate LCPP solutions when a members enters or leaves the alternatepool.

Carpooling in practice is organised hitchhiking. The driver has ultimate control over the situation but some agreement was made in advance of the journey to set the destination. Furthermore, the driver might not have checked his[1] legal obligations to the passenger in the event of an accident and the passenger takes on a safety risk in riding with a stranger. [20] shows how a driver can check their liability. The aim is to provide *reputation* and *credits* for the driver or passenger that took on the share on a voluntary basis. The reason why ordinary hitchhiking is very popular (despite the highest personal safety risks) is that overheads involved in negotiation and notification are minimal. A

[1] Male and female references are analogous

driver needs to be willing to share as well as have a sharing motive. Manufacturers of motor vehicles have demonstrated the potential of sharing. Citroën created the Osmose concept car, revealed at the Paris 2000 Motor Show, to address responsible vehicle use by design specifically for urban areas. Sharing was achieved through highly technological means and the car allows passengers to walk in and out easily. We want a good solution to cut as many steps in the negotiation as possible, to flag routes that become available, and allow maintenance of reputation and credit. The benefits would then be clear over hitchhiking and provide a safe alternative to ordinary commuter journeys, which is the aim of carpooling. Genghis can maintain itself without constant human attention, a point inspired by notes in [10]. Examples of self maintenance might be proactivity in keeping alternate pools efficient, coping with concentrated use and having ways to deal with misuse like bogus journeys.

2.2.1 Carpool Types

Three types of carpooling have been identified, two of which have been mentioned. A pooling process can have various characteristics which we have currently generalised into the all-inclusive term of carpooling. As a pretext to these types, the reader should note that a return journey is a new instance of a journey.

Standard This is an axiom of the pooling process. An instance of a journey is available, together with a certain number of other journey characteristics such as start and end points, quota, date and start/end time. A pointer is present to the driver of the journey, with multiple possible pointers to passengers.

Eventpool [29]. The idea here is that we will get a large *concentration* of pool activity (i.e. matches and search targets) in a given location or time which is due to an event like a football match. Further research will need to be undertaken to study the parameters of focused activity during events. It is possible that in certain situations, this may appear to be an alternatepool which has a single alternation. To illustrate, people going to and coming back from a football match is one alternation, which may recur every time there is a football match. If further alternations are likely to happen, then we have an alternatepool and not an eventpool.

Alternatepool Suited to a static group. The formation of a group can be proposed by Genghis or by an initiating user that invites members. The invitation process may be computed and proposed by Genghis (LCPP) or manually led by the alternatepool founder. It may be prudent to moderate group membership to ambivalent users. A

pool of this kind needs to track credits to satisfy its members, in addition to tracking user reputation - which occurs for users of all pool types. Genghis could proactively recommend members for removal or addition to an alternatepool. It might achieve this through scheduled computation of a LCPP. A further point is that an alternatepool group only takes at maximum one recurring journey. If we allow multiple journeys to be taken, we lose the ability to keep the group members optimal. Further details are found later in this study.

■ All pool types allow feedback for calculating reputation and alternatepool also has a credit log. ■

Given that Eventpool has few distinctive features, we conclude that there are *two primitive pool types*. For the three pools, user reputation indices are maintained by feedback from completed journeys. There are a number of reputation models, for example at auction sites like *eBay.com*[2], but we propose a more extensible system in Social ReGreT.

2.2.2 Character and Reputation

Social ReGreT [16] looks at social networks between individuals and uses social network analysis techniques to identify features of the society that may be important in ecommerce. We calculate reputations by constructing a *sociogram* which is a directed/undirected graph with/without weighted edges depending on the social relation. Doing this for humans is difficult due to the diversity of social relations, but multiagent systems can help to create workable sociograms. First they propose *individual dimension*, which graphs the direct interaction of two agents in an outcomes database. From this database, grounding relations are defined that link a reputation type with a list of issues. A table of these relations is provided in a later chapter. Newcomers are unable to amass a reliable reputation, prompting the authors to coin the concept of *social dimension*. We use other agents to gather three types of social reputation - witness, neighbourhood and system reputation. This excellent paper concludes with a proposal about integrating ReGreT with a negotiation model to increase the success of negotiation. We will be devoting a chapter to study and implement the ideas proposed. From first thoughts, the social dimension may be impractical to calculate in lieu of the small set of variables to maintain our reputation index. For example, the pooling tendencies of to_arrive_late, to_not_showup or to_overcharge are only affected by two things - an agent returning high costs for journey proposals (to other agents) or human feedback from journeys.

[2]eBay is a registered trademark

Restricted personal information may be available as a subset of the reputation file for a user. More personal information might become available for a confirmed journey with a user, and we will (inherently) need to allow free-form telephoning/SMS/emails between users after a journey deal is sealed. The aim is to minimise the need for this, unless specific information needs to be relayed to a participant.

2.3 The Viability of Carpooling

Although concerned with the practicalities of carpooling, we must understand why carpooling is not as widespead as it should be - a discussion where there is a lesser extent of academic literature.

Within the set of entities participating in a system, a critical number of participants in a given Driver/Passenger ratio is required to make a carpooling system *viable*. Ambivalents may be ignored for viability since they contribute to both share proposals and offerings. Without a properly scaled pooling setup, participants are prone to get into a vicious circle of failures where passengers never find a share and the system degenerates.

Many drivers - less passengers Drivers rarely get a share passenger; passengers are usually accomodated on most journey searches. This state is defined in this study as *complacent*.

Many passengers - less drivers Passengers find that the availability of share *slots* or *quota* is important for popular share routes. This state is defined as *dynamic*.

The above is of course, a crude simplification of an optimal ratio. [33] started off with the assumption that an 80% match rate would make a pooling club viable and based their subsequent analysis on Melbourne, Australia. They divided Melbourne in zones and formulated lists of acceptable trips between zones. This was represented as a matrix where indirect paths between zones/nodes were not considered. Some useful relations were identified, namely the percentage of matched trips S, given by

$$S = 100 * \frac{offersmatched + requestsmatched}{offers + requests + ambivalents} \tag{2.1}$$

It concluded with the news that a 2.5% *population participation rate would result in an 80% success rate*. The population in question is the total population of the test region.

Minimising failed matches and optimising driver usage would grow Genghis at an exponential rate as more entities realise the success of the scheme, evidenced through [21].

Ambivalents have been found to always increase participation according to [33]. I hypothesise that if Genghis was pre-loaded with a set of users and journeys before use, it would stand a better chance of success. We aim to increase the total journey dataset, and it is possible in the distant future to find interfaces to import users and journeys from other carpooling systems other than Genghis.

Contrary to claims by a large array of pooling services that show a vast improvement to commuter congestion in cities, the actual success of carpooling is severely constrained by a diversity of human and operational variables. Carpooling remains difficult to realise, given that humans (who remain unpredictable) can throw the element of unforeseen events that the system cannot absorb. The answer to this is strict control of human interaction with Genghis.

Liabilities The facility to provide a carpooling system should not result in the creators of the system liable to prosecution for crimes that result from shares initiated by the system. Users have to agree to liability statements for rides, an example of which can be seen at [19], with a sample letter for drivers to send to their insurance company [20]. In the UK, National Carshare [24] have a membership card which is exchanged when users first meet.

Datastore/Privacy Data protection is a legal requirement for a practical system, and is applicable to information like car license plates, home addresses, and personal profiles. One solution to storing a user's home, target and destination is to ask the user to *point and click* a cell on a graphical map for storing this information [10]. We note further that alternatepools can be private or public - excluding their potential journey offerings from search results if private. Later in this study, we define how user data is protected at the process level in three states - *private*, *public* and *contingent*. Contigent protection maps private to public data in a state such as one where a carpool deal has been sealed, and certain data which are normally private need to be exchanged by the two parties.

2.4 MultiAgent Systems

The dynamic carpooling problem seems to lend itself well to an application of multiagent systems. Definitions of multiagent systems are subject to argument, but some categories of software agents can be identified from [11] together with their abstract architectures.

Deductive reasoning agents These agents use symbolic or logical operators to prove formulae, facing two issues - the transduction problem and the representation

problem. A theory is encoded stating the best action to perform in any given situation.

Practical reasoning agents Practical reasoning is deliberation plus means-end reasoning, where the outputs of deliberation are intentions. A planning problem is solved (over a set of actions) and this kind of agent operates in a control cycle. Dropping intentions involves the determination of three commitment strategies - blind, single minded or open minded commitment. If the index of *dynamism* in a competitive environment is high, cautious agents are effective.

Reactive and Hybrid agents Purely reactive agents respond to new percepts, for example a thermostat. There is the promising result of *emergent intelligence*, proposed by Brooks, namely the subsumption architecture. Genghis agents will be fulfilling roles that are essentially reactive, in other words, agents will be activated truly dynamically. This wonderful idea bodes well to the concept of agents as independent, with different goals. In my opinion, the subsumption architecture realises the potential of agents in co-operative environments.

From first principles, deductive reasoning agents may be necessary for journey assembly from many journey proposals, giving users *multihop* journeys. We can choose to program these assembly procedures using appropriate rule-based tools like Drools, but it might be better to return single journey proposals first before extending to journey composition. Purely reactive agents will have other roles like replying to proposals or interfacing. We already see agents emerging with characteristics that do not belong to a class.

The benefits of using MAS are subject to argument. [9] write in their agent roadmap:

> Several observers feel that certain aspects of agents are dangerously over-hyped, and that unless this stops soon, agents will suffer a similar backlash to that experienced by the Artificial Intelligence community in the 1980's.

I disagree, since 5 years on, we have found constructive applications to MAS that have changed our toolsets of agent construction and given us solutions where algorithmic approaches are impractical. [14] provides a balanced discussion of the *needs* for agent computing in a social sciences setting and comments on equilibrium, computability and stability. Work conducted at HP Labs by Dave Cliff includes market agents [4] that trade in a virtual e-marketplace where the marketplace itself is subject to change through a genetic algorithm. The aim is to use the algorithm to automatically design new agent based marketplaces that are more efficient that those designed by humans. ZIP (Zero Intelligence Plus) artificial trading agents have consistently outperformed human traders in Continuos Double Auction marketplaces [6].

About transportation scheduling, [9] (p.295) describe:

> The domain of traffic and transport management is well suited to an agent-based approach because of its geographically distributed nature ... there are two types of agent - one representing the customers who require or who can offer transportation, and one representing stations where customers congregate in order to be picked up (when and where they want to go) and the station agent determines whether their request can be accomodated, and if so, which car they should be booked into

This seems uncannily precise in its application to this study. We will be using an approach where agents represent customers. But for the role of a station agent, we have a choice between giving each customer agent deductive powers to understand broadcasts (like in a real train station for example) or fulfilling the same purpose through dedicated broadcast and *query* agents. We should take care to note that our system may not have nodes where people congregate. The net effect of this difference might be better reasons to use deductive customer agents that act for themselves and merely *use* other agents as tools. The 2 studies cited in [9] are [2],[7] - now reviewed due to their relevance.

MAS in Traffic and Transportation [2] describes applications of MAS in traffic and transportation, beginning with an overview on AOT (Agent Oriented Techniques) and the BDI paradigm. The areas identified where MAS will have an impact are in the analysis and description of traffic systems, increasing the autonomy of traffic components and an integration framework (for example an Emergency Rescue Management centre that links up accident, pollution and decision support modules). The DASEDIS architecture is described to model traffic, based on a BDI model. We see models of agent behaviour when cars have to overtake each other and free driving. Carsharing is covered, where agents represent stations and customers. Other than these details, aspects of methodology are not covered, and this is a general overview.

DAI (Distributed Artificial Intelligence) [7] write a high level overview of distributed artificial agents in cooperative transportation scheduling, and loathe inefficient European logistics practices. They describe the MARS simulation testbed, and show how their proposal differs since the trucks have local planning. The solution to the global order scheduling problem arises from local decision-making of the agents. This is a similar idea to the subsumption architecture, but the InteRRAP agent achitecture defines an agent by a set of functional layers. There is useful detail in this paper about the specific work of agents, which I expect will influence my design. More importantly, the use of *temporal* grants/rejects should be considered or rejected for use in Genghis.

Genghis is not an intentional system. To make things as simple as possible, design should focus on benevolent agents that share tasks. In this study our concern is not merely a proposal for a given journey, but multiple partial proposals for a given journey, giving us scope to deal with multihop problems. To do this, we need to invent some kind of *cost* that is understood by the proposing and receiving agents while calling for journey proposals. The assessment of these costs from each proposal will help an agent sort a set of matches by cost amongst other things, where the lowest cost is the solution with one hop from the journey start to end point. Cost might consist of a factoring of *time/date proximity* and *location proximity*. The proposal exchange protocol is ideally the FIPA standard of the *Contract Net Protocol* (CNP) [17], which is used for negotiation where we intend to collate results after a call for proposals (CFP). Further details of a specific design solution to this problem are given in later chapters with a domain ontology. Development phases are discussed in Section 2.5.2.

We end this section by noting the requirements of agreement and negotiation. Agents reaching agreements with each other have desirable properties, namely *convergence*, *pareto-efficiency*, *individual rationality*, *simplicity* and *social welfare*. A **negotiation setting** must have four components [11] - A negotiation set (possible proposals agents can make), a protocol (reliant on a common ontology), private strategies (necessary only when agents are competitive) and termination (a rule that determines when a deal is struck). Although we cover negotiation here, it may turn out that agents never negotiate, but propose their best offering, which can be likened to the first step of the Zeuthen protocol. After all, an exchange of Agent Communication Language (ACL) messages only occurs if an agent is able to satisfy a CFP within cost bounds.

FIPA To lead into the next Section, we identify that FIPA has defined a set of standards to create multi agent systems that are interoperable. Specifically, FIPA has specified the ACL and identified three mandatory roles. The Agent Management System (AMS) handles platform access and agent registration. The Agent Communication Channel (ACC) provides for communication links, and the Directory Facilitator (DF) provides a yellow pages service. Jade conforms to FIPA requirements.

2.5 Development

2.5.1 *Jade*

Our choice of implementation options for this system is diverse. The prescribed choice of Jade arises from its widespread use, adherence to FIPA standards and a skills match

(in Java).

Jade is a distributed agent platform, which can run on several hosts, each of them executing a Java Virtual Machine. It is FIPA compliant, which includes an AMS, the Directory Facilitator (DF) and Agent Communication Channel. Jade uses one thread per agent instead of one thread per behaviour, so that a scheduler can carry out a round robin policy among all the behaviours available in the queue.

Interfaces to the web from Jade are a final goal of this study, and native scripting languages like JSP (Java Server Pages) would be implemented as servlets through a ProxyAgent. The ProxyAgent is an interaction gateway to separate Jade and the web application. We may also find it better to use many ProxyAgents.

It is unlikely that we would use Institution Definition Languages or any mappings from other platforms to create prototype institutions. Unless the desire arises to interface multiple Genghis installations, the inter-platform Message Transport Protocols need not be extended.

2.5.2 Development Methodologies

The means by which software is engineered is a large field of study, and developing MAS calls for a special range of software development phases or so called methodologies.

For development, we consider extending object-oriented methodologies. Software development with JADE supports this notion since Java is object-oriented. Some methodologies are shown below [11].

The AIII Methodology Here the focus is on production of internal and external models. The external model presents a system level view of agents and the relationships between them, and the internal model is entirely concerned with the beliefs, desires and intentions of an agent. The external/system model develops in a way similar to that of an object oriented hierarchy.

Agent UML We note from the outset that UML for object oriented systems is not a methodology, but a language for documenting models of systems. Associated with UML is the methodology known as the Rational Unified Process. UML is a very widely used standard, but that does not mean it can immediately map to MAS. Additional support for expressing concurrent threads and rich modelling of *roles* is being evaluated by the Object Management Group (OMG) and Foundation for Intelligent Physical Agents (FIPA).

Cassiopeia This methodology comes from the school of knowledge engineering techniques and not object oriented approaches. In contrast to Gaia and AIII, this is a

bottom up approach. One starts with the identifying behaviours required to carry out a task, moving through to relating these behaviours and then organising them as groups.

The Gaia methodology[3] lets us go systematically from a statement of requirements to a design good enough to be implemented. In applying Gaia, an analyst moves from abstract to increasingly concrete concepts. Each successive move shrinks the space of possible systems that could be implemented. Table A.1 shows abstract and concrete concepts in Gaia. Appendix A details other points about Gaia.

In Gaia, an understanding of the system is captured in the system's organisation, that is viewed as a collection of roles which stand in certain relationships to one another. A role is defined by four attributes - responsibilities, permissions, activities and protocols. Responsibilities determine functionality, and are a key attribute in a role. There are further details to this methodology from [34] which will emerge naturally as we specify Genghis. We will use the practical experience from [13] to help us through this process. [13] walks through the engineering of a MAS implemented with a JADE framework.

2.5.3 Mapping and graphics

It is preferable to use simple array-like stores of grid cells, and overlay these with graphical information that makes it useful.

The availability of graphical maps is vital to users of Genghis since this is a key element that makes the system work in practice. Popular web-based mapping services provide facilities for developers to tap into their APIs for mapping in their applications. Multimap.com[4] is one of the big players in web-based mapping, primarily allowing users to generate driving directions to any location in the UK, as well as showing up hotels and services in the area. I have secured a developer account with Multimap, which uses an XML driven API [22] to provide route graphics given certain parameters. One of the commercial applications demonstrated by Multimap was that of the Shell Petroleum Company - where a driver's origin and target area are fed into Multimap which displays the route graphically and highlights Shell fuel stations *along the route*. If this database, remotely hosted at Multimap was dynamically fed location start/finish information from Genghis search results, it could be used to recommend and display physical routes, as opposed to straight or curved line representations of a journey. This approach is unfavourable since Multimap can withdraw their services at any time.

[3]From the famous hypothesis about world ecosystems by James Lovelock

[4]A trademark

Existing solutions [10] integrate a commercial GIS for Intel Ireland into their system, namely Transcad GIS to handle geographic data and Autodesk Whip as a plugin for pushing mapping data on the user desktop.

Studies into GIS have yielded fruitful results. *Mapgets* is the name given to *map widgets* proposed by [1] for the querying and manipulation of geographic data. These widgets provide a means to manipulate geographic data interactively in a new kind of GDUI (Geographic Database User Interface). Their application to this study is unclear as yet.

Complexity of the GIS will determine **how** we match routes and calculate a solution. These two cases show the difference in the way a GIS can match routes.

Node-Node In a simple grid of cells, we use either node to node or coordinate to coordinate route matching. This may be a straight line from point to point. Intermediate obstacles are ignored. This grid of coordinates has to be visible and useful, so one simple approach is to use this grid in the GUI, covered by a mapped set of images that describe the terrain and features to a human.

Routing algorithms Given a set of geographic data or obstacles described in their own way (for example with [1]), we determine a *set of routes* to another node, from which we choose one route. This approach is used in [29] to maximise dropoffs/pickups along the way. We may also specify we want the shortest route or fastest route, and these could possibly be NP complete problems. Fortunately, in most cases it seems that *brute force* listing of all routes to make our choice may not list very many options. This could be why online mapping services like Multimap seem to be able to handle a given load of queries, but I don't have information to make such a hypothesis.

It appears that our choice can evolve from Node-Node to routing algorithms. We will not focus on picking up and dropping off people along a route. A request to go from A to B and scoring matches or partial matches through a proposal to a CFP is a sufficient beginning.

Chapter 3

Carpooling Processes

In this chapter, we formalise the domain and define an ontology with process definitions.

Ontology We construct an ontology description for Genghis based on pool types. A framework of discrete notions lets us set down the rules that govern a model of the domain. This makes sure that our agents can transfer domain knowledge without conflicts.

Pooling definitions Carpooling split into two problems that Genghis will address. Both problems have synergies in that they use the same ontology, but will have alternate goals with the dataset.

Data storage A.3 extends this chapter by providing a schema to serialise and rebuild agent information.

3.1 Carpooling Ontology

Standard JADE Agents like the DF and AMS agents communicate by using the `FIPA-SL0` content language with the `fipa-agent-management` ontology. Hence we should build an ontology for other agents in this domain.

An ontology is a formal description of the relevant parts of the domain. We use it to share common understanding on the structure of information between agents, and to separate domain knowledge from operational knowledge. It consists of a set of classes (or concepts) and a set of properties (or slots). The same concept may be in several independent classes.

Protege-2000 is used to create the ontology, and we are able to export the Java beans for use with Jade using the Ontology Bean Generator plugin. Our root class is a Concept, which has subclass AgentAction. This is the common ancestor of all actions in an ontology such as requests and proposals, so we nest our actions below this node. We will see that ACL Messages to agents are wrapped in an Action (`jade.content.onto.basic.Action`).

For the sake of brevity, ontology HTML documentation is on the CD-ROM. There is similarity to the database schema in A.1. The ontology started with a basic design and evolved as we realised things about the domain and MAS (like the descendants of AgentAction).

■ `/DISSERTATIONJADE/REFERENCE/ontology/index.html` ■

■ Java beans at `/DISSERTATIONJADE/src/genghis/ontology/` ■

Concept User There is no User since we have assumed that the agent AIDs will correspond directly to the Users.UserID primary key. From this primary key - we consult the database to retrieve user attributes. When Genghis is started, it is important that our UserAgents are numbered according to how many records the Users table contains. We name our agents with the prefix `user` followed by a number which is the primary key of the record.

MyCostedJourney An object which is used to despatch the response to a CFP after the agent has made its calculations (subcontracted to the DecisionModule). A CFPJourneyResponse is a multiple of these.

CFPJourneyRequest If we allow the user to specify more parameters, like the importance of time or starting location to their journey request, then items (or slots) should be appended to this Action concept.

3.2 Formulaic Processes

Here we define the nature of the two primitive pool types. Each of these types is called a formulaic process, and this study aims to implement one or more of these. In alternatepool groups, we aim to foster a sense of community. Standard pools do not have underlying groups. This project implements the Standard type onto Jade.

3.2.1 Standard

Standard pool S holds members of instance User.

The following data are maintained.

Members of S.
A measure of reputation for a member of S.
A set J of instances from Journey. A member of J is j.

The following are true.

There is at least and never more than one member d from S at any time who has the role of Driver.
There is at least one member g other than d in S.
Members of S\d have roles Ambivalent or Passenger.

A single process for driving or being a passenger for this pool is defined like so.

Member from S chooses to drive as d or be a passenger as g. d or g associates itself with j and completes j. d or g can choose to provide feedback on any member associated with j.

Addition of members Users outside S choose to apply to join directly and are added automatically if the quota of j is not exceeded and certain other conditions are met.

Removal of members d may choose to remove a member of S, but the removal of d would terminate S. A member g from S may remove itself.

Reputation growth Members of S can provide feedback on all members of S but themselves.

3.2.2 Alternatepool

Alternatepools are carried out inside a group. Ideally, all Journeys classed in this type should have a return Journey, but sometimes that will not happen so we do not enforce it.

There is an Alternatepool Z. It holds members who are a subset of G which holds instances of User.

The following data are maintained.

Members of Z and G.
Attributes to describe G to non-members.

A measure of reputation for a member of G.

A log of journeys which have been completed by a member of G. Credits are calculated by adding the number of journeys driven by the member and subtracting the journeys where the member was a passenger.

A set J of instances from Journey. A member of J is j.

The following are true.

There is at least two members in Z.
One member of Z is chosen to be a Driver.
There is at least two members in G.
Members of G have Ambivalent roles only.
There is one member f in G called the founder.
Member f moderates G.
The credit of all members of G is visible to members of G.

A single process is defined like so.

Member x from Z chooses to drive or be a passenger. x associates itself with j and completes j. The log of journeys by x is appended with an entry. x can choose to provide feedback on any member associated with j.

Moderation Potentials awaiting moderation cannot partake in j associated to any member of G. f chooses which users will join G from a list of applicants. Approval by f gives the applicant membership to G.

Addition of group members Non-members of G apply to join directly. Genghis may request f to join a user to G. Both cases result in new members entering moderation.

Addition of alternatepool members Members of G apply to be members of Z if the quota for j is not exceeded and certain conditions are met. Genghis may also propose or execute membership to Z on behalf of members of G.

Removal of group members f may remove a member of G, but the removal of f would terminate G. f may surrender its role as founder to another member y from G, resulting in y renamed to f. Member y from G may remove itself.

Removal of alternatepool members A member x from Z may remove itself. No member of Z\f may remove any other member of Z.

Reputation growth Members of Z can provide feedback on all members of Z but themselves.

Chapter 4

MAS Development

We describe Genghis as a MAS. We think about the basic agent interactions. We formally extend this Chapter in Appendix A.

Design The Gaia Methodology is used to tabulate concepts in a step by step way, ending up with a concrete description of agents. This is the skeleton of a design without algorithms or deductive steps.

Interactive Processes The general processes that occur are illustrated. This section does not include deductive steps for the two types of car pooling, since these deductions (like route matching or group clustering) are infinitely configurable in better ways.

4.1 Agents Design

Our first assumption is that agents in this MAS are eternally alive, whether or not the human user is logged in. For every user, there is a corresponding UserAgent representing them at all times.

The standard way to negotiate which is noted in this design is the Contract Net Protocol shown in Figure 4.1. The agent definitions achieved through the Gaia process are inserted into A.2. We create the following agent roles.

UserAgent Represents a human user and their allowed interaction with Genghis.

ProxyAgentN This agent will service HTTP requests and act as the middleware between a Jade container and web application. There may be N different agents characterised by this role.

JourneyRoundupAgent Flags journeys which are past their end time for human feedback.

JourneyNotifyAgent Keeps watch on the wanted and active journeys and flags User-Agents if any match comes about. The idea here is that the human is notified of new match score and can look into seeing the particular option matched. It may choose to rank this match within a bound and give it a traffic light signal, actioning it only if the light was green. The act of booking a match that was previously wanted reverts to the standard booking process.

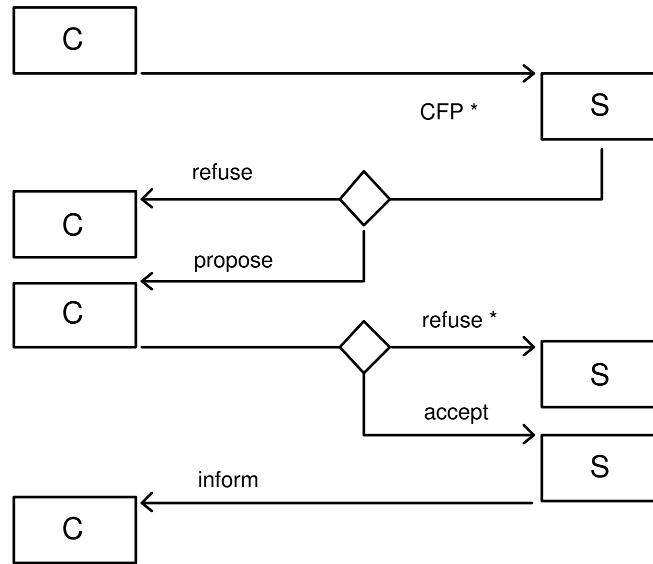

Figure 4.1: Basic Contract Net Protocol, implemented in package jade.proto

The ontology and database schema are a necessary part of the system. Notes on the schema are provided in Appendix A. A list of illocutions for each agent is not given since it is apparent from the roles and the Contract Net Protocol which performatives each ACLMessage does contain. Even so, one could if they wish make a list of illocutions for each agent in the form *performative(sender, receiver: agentAction(parameters))*.

4.2 Temporal Connectives

The idea presented in [7] is not appropriate in our flavour of the Contract Net Protocol. Temporal connectives are an extended logic from the simple refuse/propose we know, which have operators to express *temporal grant* or *temporal reject* to proposals.

If we allow agreement based on temporal connectives, the concept of a cyclic agent watching the active and wanted journey datasets becomes obsolete. We impose additional behaviour into the UserAgents who need to cyclically service these temporal grants and rejects as their own behaviours. The efficiency of a single agent that runs through the database and informs UserAgents is lost.

Unlike the example previously quoted about a station master making announcements to everyone (who can understand them), it is simpler to use the standard fields in a journey to inform the UserAgents of a given record that a search filter has been updated. We invent the concept of a *match score*, which the UserAgent crudely interprets as a traffic light signal by slotting the score into bounds. The idea is that when the wanted journey (match score) goes into the green bound, the UserAgent is informed. Whereas - temporally accepting a proposal or having the UserAgent run its own monitor to wanted journeys is over-complicated.

4.3 Basic Requirements

This Section will introduce interactions that would occur in Genghis. They refer to the Schema and Roles that we have identified. The items below have been used as the basis for detailed design information in Appendix A.

Adding a new journey The simple process of adding details of a journey, requested by Ambivalent or Driver. A UserAgent inserts a confirmed journey that they are *driving* into the active table of journeys. (Figure 4.2).

Querying for a journey The UserAgent provides data for an explicit CFP (Call for Proposals) for a journey. The responses are collated by cost options. The CNP (Figure 4.1) is used to administer negotiation (Figure 4.3).

UserAgent - seal the deal Also known as the booking process. Interaction depends on one of two journey types confirmed. UserAgent confirms an available space in a particular journey, whether is it a standard or alternatepool journey. If it is a *standard journey*, UserAgent validates input and adds a participant to that journey. In the case of an *alternatepool journey*, group membership is first checked before participation is confirmed. (Figure 4.4). Both deals can be sealed by obeying the laws of the forumalaic process (Section 3.2).

UserAgent - add wanted journey The process would happen following a query which was not fruitful or it can be started directly. A UserAgent logs a wanted journey into the Table of Journeys and an association with it in the associations Table. For the future, JourneyNotifyAgent implicitly takes up the responsibility of letting

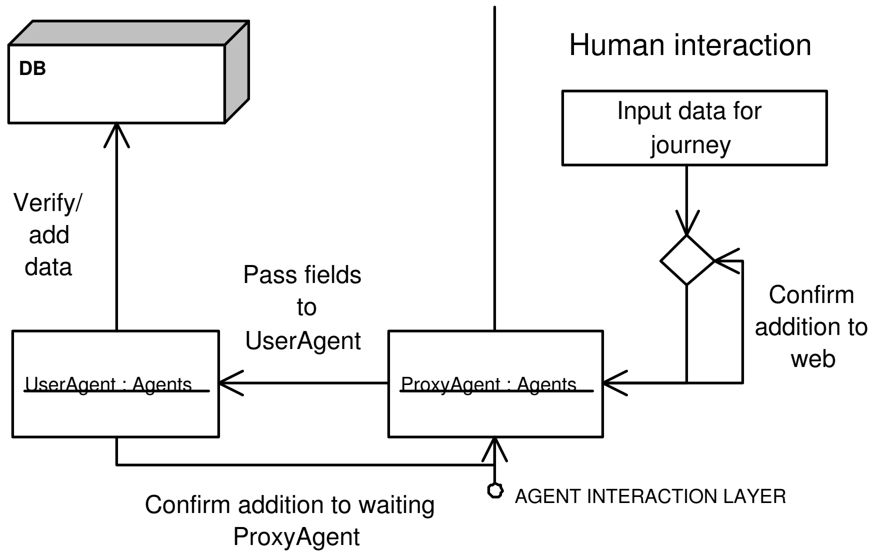

Figure 4.2: A human driver or ambivalent adding a new journey

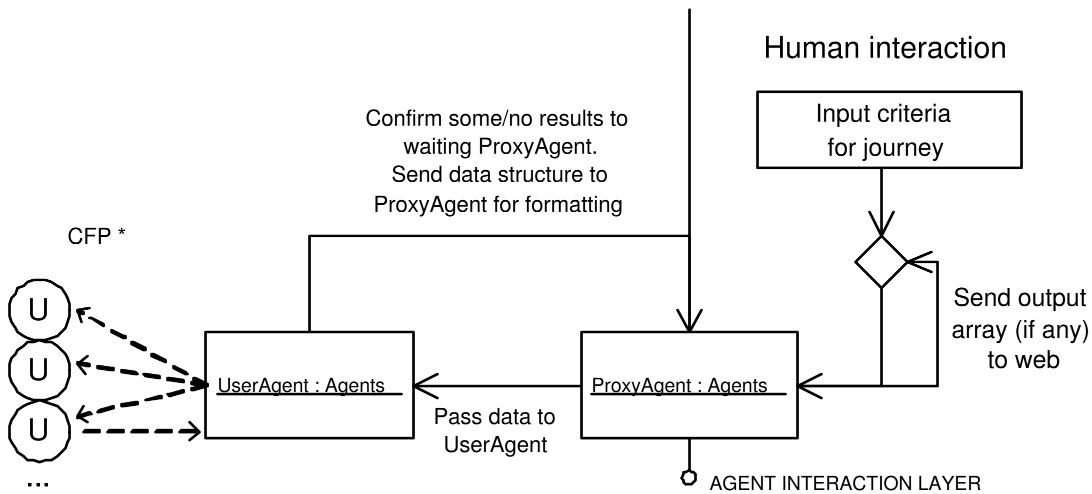

Figure 4.3: Human requesting a search for a journey - Genghis uses CFP

UserAgent know when a match is achieved. Other than with ProxyAgentN, no agent interaction occurs.

JourneyNotifyAgent - inform match Match-making is achieved through the cyclic behaviour of this agent whose role is to compare the wanted and active set of journeys. A UserAgent is told of a green traffic light bound being reached using

Figure 4.4: Human confirming participation in a journey

the bounds programmed into the JourneyNotifyAgent. An inform message is sent to UserAgent with the green traffic light status, which would let a UserAgent take further action like send an email to request booking (seal the deal). The UserAgent may not directly query this agent, and must only retrieve the latest indicator from the database if it wants to do so. This allows the JourneyNotifyAgent to work cyclically at its own pace. (Figure 4.5).

JourneyRoundupAgent A cyclic maintenance behaviour that finds out which journey is complete and transitioning journey state from *Active* to *Complete*. It logs an entry in the database to indicate journey feedback has fallen due. When the human eventually enters feedback, the UserAgent should delete the referring *pending feedback* row. The JourneyRoundupAgent interacts only with the database.

ProxyAgentN The ProxyAgent is the gateway to the web user from the Jade containers. If we need more proxy behaviours, we can create more agents, where N is substituted for a number. It queries a given agent and awaits a response from it, which is delivered in HTML to the waiting web user. As HTTP is stateless, we create and destroy one shot behaviours to service HTTP requests. This agent might have a set of activities which depends on the GUI, explored in Chapter 6, and as such is not a requirement. These agents are configurable and expandable to instruct Jade-side agents as they see fit. We also hope that each data passed by the web user is first validated before being allowed to live on Jade containers inside ACL Messages.

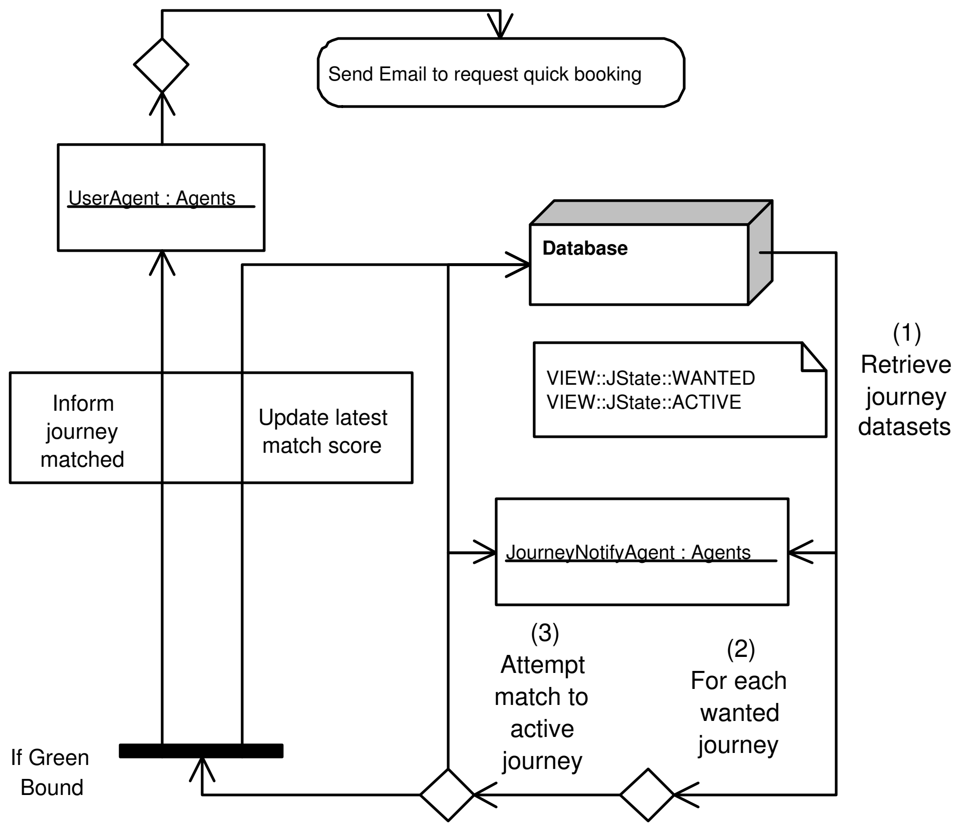

Figure 4.5: JourneyNotifyAgent flagging matching journeys to UserAgents

Chapter 5

Single Journey Matching

In this chapter we describe an algorithm to cost single route proposals for use internally by an agent. This deduces a costs for relevant journeys which we use to reply to CFP (Call for Proposals). We do not cover the optimal composition of journey responses on the receiving side. Journey composition helps a human user by showing the best sequence of journeys for their requirement, otherwise called a multihop journey. The process of working through this algorithm is emulated in a Java Object *DecisionModule* that is instantiated with arguments at run-time.

5.1 Overview

The alternatepool is not be implemented, but the essential details are given in Section 3.2. The alternatepool may use the algorithm we give here, but it may also need other *essential* modules like a solution builder for the LCPP (Long Term Carpooling Problem). None of this applies to our implementation, since we work with a view to implement standard pool only.

For efficiency our algorithm checks a given journey to the required journey in a hierarchy of 4 Levels, starting at Level 1. The current journey may be rejected at any point as a possible result.

We introduce the following notation to be used through this Chapter.

Each UserAgent has a set of journeys which are retrieved from the schema with condition

■ `Users.UserID = JourneysDB.UserID` ■

which we denote with JOURNEYS_M.

The journey which we want to try and satisfy is received from a CFP, and is denoted

by W_J - the *wanted* journey.

An agent will retrieve the set JOURNEYS_M. A member of this set is denoted W_M or the *current* journey. Our algorithm picks a W_M only once and continues checking W_M until JOURNEYS_M does not have any unchecked journeys.

The result of this entire process is the reply that we give to a CFP. The result is a set RESULTS_J, which can be empty. The number of items in RESULTS_J does not exceed 5. This ensures that the proposing agent receives a reasonable number of proposals back from all agents that can satisfy the CFP to some extent.

5.1.1 Contents of Wanted Journey

The CFP sends us W_J which contains a Journey that somebody wants. This has the standard attributes as documented in the Journey Concept. In addition, the CFP packet contains an importance weighting for time flexibility. In other words, the human also supplies how important it is that they get a journey as close to their desired start time as possible. Our maximum weighting in the algorithm for time is 30%. Hence full importance to time will result in the full possible weighting (30%) given to it in the W_J object.

We will denote the personal time importance supplied as UsertimeImportance in the algorithm that follows whose bounds are (0.0 < UsertimeImportance < 1.0). The default is 1.0 in implementation.

5.2 Decision and Cost Process

Our constraint algorithm is divided into 4 steps and we process each W_M from Level 1. The END statement at any step signifies that we restart the whole process with a new W_M until we have checked all W_M in JOURNEYS_M. We process the time importance and prepare our adjusted weighting for two more components deduced in Level 4.

We have static constants processed in this order.

The maximum time flexibility for any journey
int t = 300 minutes
The constant distance length percentage used in Level 3
int PCTG = 30 (1 < PCTG < 100)
The constant degrees bound used in Level 4
float DEGbound = 35 (1 < DEGbound < 90)
We find proportional time importance

int timebound = 0.30 X UsertimeImportance
Find out the remaining portion
int remainingweight = 1 - timebound
Use remaining portion for concepts in Level 4
int COSTWEIGHT1 = 0.6 X remainingweight (0.0 < COSTWEIGHT1 < 1.0)
int COSTWEIGHT2 = 0.4 X remainingweight (0.0 < COSTWEIGHT2 < 1.0)

There are four levels and processing of W_M, starting from Level 1 and we may proceed to the next level depending on pass/fail.

Level 1 - Sanity Check *Type Pass/Fail.*

Level 2 - User supplied weighting *Type Pass/Fail.* Time Constraints

Level 3 - Calculate relevance as vectors *Type Pass/Fail.* Geographical Constraints

Level 4 - Costing *Type Cost/Rate.* Calculate weighted costs and merge into a single reportable number, add to RESULTS_J

The feasibility of a journey is evaluated in Levels 1, 2 and 3. If it remains feasible, we use Level 4 to cost it.

5.2.1 Level 1

Is W_M of JState `Active`? If FALSE, END.

5.2.2 Level 2

$$|StartTime(W_M) - StartTime(W_J)| < t$$

If FALSE, END. ELSE we save a variable.

$$timedifference = \frac{|StartTime(W_M) - StartTime(W_J)|}{t}$$

5.2.3 Level 3

These are crucial tests we want to make for comparing vectors. Each test focuses on basic things that we consider to be a journey which might be reasonable. We know that it is

rare to have two vectors (journeys) that are coplanar since either the start/end points are unreasonable, or the direction is not good enough to earn a place in RESULTS$_J$.

(A AND B AND C) must evaluate to TRUE. If FALSE, END.

A

$$startdiffINmetres(W_M, W_J) < distanceINmetres(PCTG\%, max(W_J, W_M))$$

Find out if the distance from start point of current journey is more than PCTG% away from start point of the longer of the two journeys. Confirms if the start point of the current journey is too relatively far from the wanted journey, and is essentially a circular bounded region around the start point. If one of the journeys is a very long one, the real world catchment radius increases. Both functions convert latitude and longitude to distances in the real world, which is discussed in Section 6.1.

B

$$endingdiffINmetres(W_M, W_J) < distanceINmetres(PCTG\%, max(W_J, W_M))$$

Similarly, find out if the distance from the end point of current journey is more than PCTG% away from end point of the longer of the two journeys. We have to consider end points because we are trying to satisfy the wanted journey in one reasonable hop since the multihop journey needs a different, sophisticated algorithm.

C

$$\arccos\left(\frac{f(W_M).f(W_J)}{|f(W_M)||f(W_J)|}\right) < DEGbound$$

Find out if the current journey is *directed* somewhat towards the end of the wanted journey. We use the basic vector method of scalar product to find the angle between two vectors. This needs to fall below a bound. The function f deduces the required parts from W$_J$ and W$_M$ to return a usable vector object.

5.2.4 Level 4

We calculate costs through two concepts. Concept 1 looks at how much detour W$_M$ has to make to satisfy W$_J$. Concept 2 examines the proportion of W$_J$ which is serviceable.

1. **Concept 1** We find the round trip distance which is required to satisfy W$_J$. There is three vectors to add up the round trip distance using the Pythagorean

theorem for each vector, an example is shown in Figure 5.1. This tells us the ratio of the *detour* that W_M would make. Of course, if W_M was a far shorter journey than W_J then this journey would probably serve only as a partial match, which is a proportionality score assessed by Concept 2. In any case, we add up the round trip distance (RTD) that W_M must cover to satisfy W_J and itself.

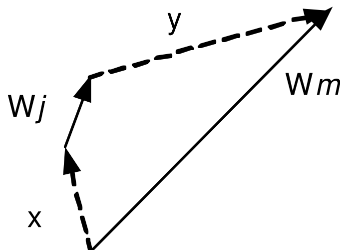

Figure 5.1: Three parts which add up the distance of a round trip

$$concept1 = COSTWEIGHT1\left(1 - \left(\frac{1}{RTD/length(W_M)}\right)\right)$$

concept1 is normalised, inversed and weighted by COSTWEIGHT1, since ratios with less difference between round trip distance and W_M are more favourable.

Concept 2 The component of a vector in the direction of another vector. This gives us projection, essentially the proportion coverage. Our aim is to find the component of the vector W_M in the direction of vector W_J. We assume that we have extracted the relevant parts from Journeys to make them vectors. Find unit vector v.

$$v = \frac{W_J}{|W_J|}$$

The component of W_M in the direction of W_J.

$$component = W_M.v$$

We then normalise this value. The intention is that W_J which are fully satisfied score higher than W_J that are partially projected. This difference will happen because W_J can be much smaller or much bigger than W_M. If it is much smaller, then we expect that it is fully satisfied, and it is much bigger, then we expect that the projection will be a fraction of the length of W_J.

$$concept2 = COSTWEIGHT2\left(1 - \left(\frac{1}{component/length(W_J)}\right)\right)$$

2. We combine items calculated so far to write the final cost.

$$cost = 1000 \times ((timebound \times timedifference) + concept1 + concept2)$$

3. We have finished. Add (RESULTS$_J$, (int) cost, JourneyID)

5.3 Return Results

If it is non empty, sort RESULTS$_J$ by ascending cost. We take entries and return these JourneyID references as our response to the CFP. Since all UserAgents will use the same algorithm, the Initiator receiving proposals can make like for like comparison and execute the job of merging received lists and sorting to present matches of least cost.

If RESULTS$_J$ is empty, an ACL REFUSE message is sent to the CFP Initiator, according to Figure 4.1.

5.4 Implementing the DecisionModule Object

The DecisionModule Java Class is passed two Journeys - the criteria and current journey. After instantiating an object of this class with two journeys, we call methods on it. It works through all the above algorithm and allows us to upgrade the logic in future. This object can also extend the APIs of rule-based packages like Drools in future. The module is a roughly similar implementation of the algorithm proposed above without Level 4.

■ `/DISSERTATIONJADE/src/genghis/DecisionModule.java` ■

5.5 Examples

Here we run through example scenarios of a pair of vectors W$_M$ and W$_J$. In these scenarios, we will assume that time difference between the wanted journey and current journey is fixed at *1 minute* and not consider it.

5.5.1 Level 3 Example

We calculate A, B, C for examples with W$_J$ remaining fixed. Figure 5.2 shows 3 examples where W$_J$ is fixed and we vary the current journey. We use simple integer co-ordinates,

but the case in point is the same for floats. Scenario 1 has overlaid on it the idea of Equations A and B in Level 2, while Scenario 2 has overlaid the idea of *reasonable* direction. Scenario C has no overlay and is a example of a journey that looks reasonably *good*, through simple observation.

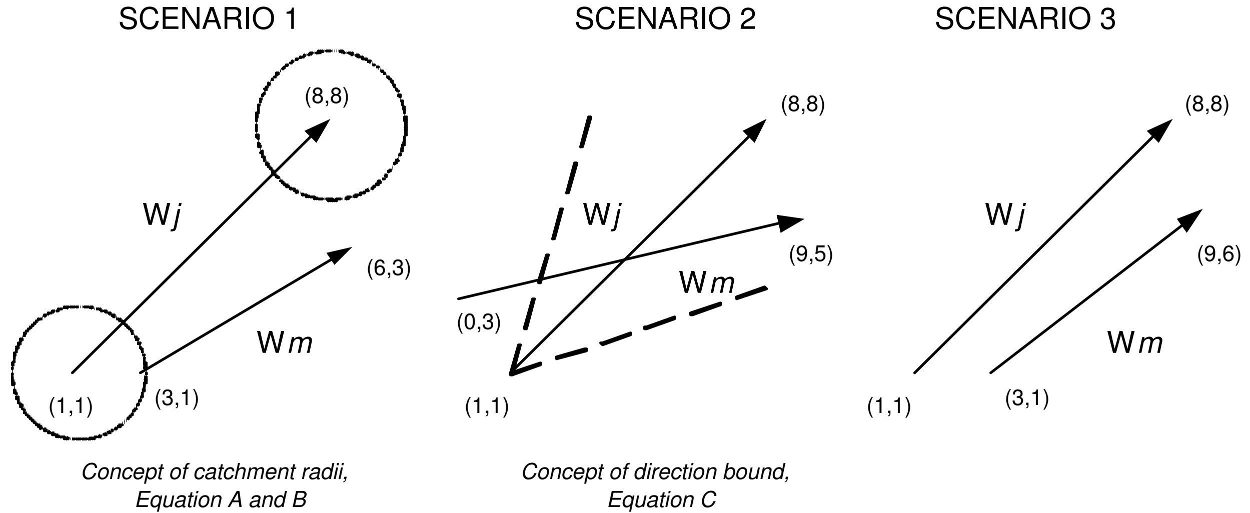

Figure 5.2: Three examples scenarios for calculating Level 3

We shall calculate results of Equations A, B, C for each of the scenarios below to find out pass/fail and make a conclusion. We shall take co-ordinate distance as a substitute for real distances, but real distances are calculated as shown in Section 6.1.1.

Scenario 1

$$(A) 2 < (30\%, 9.899) : TRUE$$

$$(B) 5.385 < (30\%, 9.899) : FALSE$$

$$(C) \arccos\left(\frac{21+14}{(9.899)(3.606)}\right) < 35 : TRUE$$

Clearly the end point is too far away to satisfy a one hop journey constraint, so we FAIL.

Scenario 2

$$(A) 2.236 < (30\%, 9.899) : TRUE$$

$$(B) 3.1627 < (30\%, 9.899) : FALSE$$

$$(C) \arccos\left(\frac{63 + 14}{(9.899)(9.220)}\right) < 35 : TRUE$$

Unfortunately, B is only slightly out, but C *just about* passes. The result is a FAIL.

Scenario 3

$$(A) 2 < (30\%, 9.899) : TRUE$$

$$(B) 2.236 < (30\%, 9.899) : TRUE$$

$$(C) \arccos\left(\frac{42 + 35}{(9.899)(7.810)}\right) < 35 : TRUE$$

We can see that it looks reasonable, and C has a single digit value around 5 degrees so it does PASS.

The above shows that our Level 3 makes intuitive sense at the very least. The parameter constraints can be changed if so desired.

5.5.2 Level 4 Testing

This is a difficult Level to test, and examples will only prove *intuitive* hypotheses. There is no open way to solve this problem to perfection, and the concept sketches have been created after a great deal of working out through example cases. It is best to test this Level on future iterations of Genghis upgrades. The implementation for now, returns fixed costs for suitable journeys that passed Level 3.

Level 4 is a best effort, which looks like it should work, but there is many other things to consider for scenarios that involve costing, and fine-tuning needs resources.

Therefore, the coded version fills RESULTS_J with suitable journeys that passed Levels 1, 2 and 3.

5.6 Coding

Many methods for this work on Level 1 through to 3 are to be found in Java libraries. Especially useful are vector algebra APIs, and any credits due are in the DecisionModule source code.

Level 4 is not coded and does not therefore calculate a final cost from DecisionModule. This is because it needs much more extensive proof of concept and is the largest subproject in coding this algorithm.

This simple algorithm is to be used by each UserAgent, tackling the essential points of a journey that might be useful. If we write something elaborate it is likely that matches that were good by inspection will not pass through the tests, and proofs become necessary. It is helpful and co-operative to return costed partial matches than to withhold any reasonable journeys.

The complexity of this algorithm is linearly dependent upon the number of items in JOURNEYS_M.

Chapter 6

Web interaction and Mapping Overlays

This chapter discusses the recognition of a location in global terms, as well as the conceptual role of the ProxyAgent in performing complex web interactions with the human user and Jade container.

Latitude and Longitude A brief look at how we describe journey points.

ProxyAgent How the interaction is achieved through a JSP/JADE interaction layer. The idea of ProxyAgent and the decision not to implement due to command line arguments being available.

6.1 Latitude and Longitude

Various standards exist for the expression of latitude and longitude, which traditionally expressed in degrees, minutes and seconds.

Cartographic products on paper usually do not use decimal notation, but conversion for use in databases is common. The ISO have created a standard for this information called ISO6709, and the following is quoted from a summary of the paper.

```
ISO6709:1983 "Standard representation of latitude, longitude and altitude
for geographic point locations" is a format designed for usage in human
readable compact file formats, protocols, etc.

The British Equivalent is BS 5249-3:1983. The standard allows both a
```

```
minute/second representation as well as a decimal fraction representation.

Latitude can be represented as

  DD.DD          degrees and decimal degrees
  DDMM.MMM       degrees, minutes and decimal minutes
  DDMMSS.SS      degrees, minutes, seconds and decimal seconds

Prefix with + north of and on equator, and with - south of equator.

Likewise, longitude can be represented as

  DDD.DD          degrees and decimal degrees
  DDDMM.MMM       degrees, minutes, and decimal minutes
  DDDMMSS.SS      degrees, minutes, seconds, and decimal seconds

Prefix with + east of and on prime meridian (Greenwich), and with -
west of Greenwich up to the 180th meridian.

Leading zeros are required for latitude and longitude.

Optionally, append altitude in meters (prefix with + above and on the
geodetic reference datum and with - below it).
```

We have used double types as our database representation of the two global geographic indicators. The Java type on grid locations the ontology is a float. Both of these are fine, since we know that mySQL has problems working with float types when an SQL query returns zero rows.

We concern ourselves only with describing a location, not its altitude. To achieve accuracy in our locations, we have to *fix* a minimum accuracy. All Journeys should store data to this accuracy, but we cannot enforce it because the GUI might feed a range of decimal places as input.

Consider the following example using the online mapping of MultiMap.

```
BATH, UNITED KINGDOM
lat: 51:22:33N (51.3758) lon: 2:21:36W (-2.3599)
http://www.multimap.com/map/browse.cgi?lat=51.3758&lon=-2.3599&scale=100000
```

It is therefore possible to readily link visualise links *inline* with a HTML GUI. A number of factors can be customised through the direct URL service. We notice in particular

that the best resolution achievable with MultiMap calls for a maximum of 4 decimal places in the latitude and longitude expression. Our decimal types - double and float are adequate to store numbers at this level of accuracy.

Furthermore, MultiMap provide an XML API which we can use in two ways.

Visualisation Aid As shown above, a location may either be visualised or be *text-searched*, then keyed in manually by the human with its latitude/longitude as reported by the web service.

Visualisation to Storage As we discussed in Section 2.5.3, we wish to allow search/display of locations so that target co-ordinates can be understood by the human user. It is possible to use the MultiMap API to achieve this by passing values from the GUI into their decimalised format for use by the Jade side.

Of course, the use of the API relies on a front end to Jade with the basic features to be created first, and furthermore, agreement from the company on the continual use of their service.

6.1.1 Conversion to Distance

Here we show Java procedures using the Great Circle Method to convert the latitude/longitude difference to real distances. For very long journeys, this may be less accurate due to the small curvature in the surface of the Earth, which is technically an oblique spheroid. Extreme altitude difference can also affect journey distance.

```
/*
    South latitudes are negative, east longitudes are positive.
    lat1, lon1 = Latitude and Longitude of point 1
    lat2, lon2 = Latitude and Longitude of point 2
    unit = the unit for results
       where: M is statute miles
              K is kilometres (default)
              N is nautical miles
*/

private double distance(double lat1, double lon1,
double lat2, double lon2, String unit) {
  double theta = lon1 - lon2;
  double dist = Math.sin(deg2rad(lat1)) * Math.sin(deg2rad(lat2)) +
```

```
    Math.cos(deg2rad(lat1)) * Math.cos(deg2rad(lat2)) *
    Math.cos(deg2rad(theta));
  dist = Math.acos(dist);
  dist = rad2deg(dist);
  dist = dist * 60 * 1.1515;
  if (unit.equals("K")) {
        dist = dist * 1.609344;
  } else if (unit.equals("N")) {
        dist = dist * 0.8684;
        }
  return (dist);
}

//degrees to radians
private double deg2rad(double deg) {
  return (deg * Math.PI / 180.0);
}

//radians to degrees
private double rad2deg(double rad) {
  return (rad * 180.0 / Math.PI);
}

//example
system.println(
distance(32.9697, -96.80322, 29.46786, -98.53506, "M")
+ " Miles\n"
);
```

A GUI should use Miles for British users.

6.2 ProxyAgents

The package `jade.tools` contains a simple agent called a SocketProxyAgent. It is a bidirectional gateway between a JADE platform and an ordinary TCP/IP connection. It converts and parses ACL Messages as simple ASCII strings. This is the first and most basic level of web interaction we have available. We have not implemented JSP interfaces to Genghis due to lack of resources, but the process is straightforward.

The main point of interfaces is that we can get data into the Jade system. We can already achieve this through the command line, with ProxyAgent/JSP being just a better wrapper, as demonstrated below.

```
java jade.Boot user1:UserAgent(1 arg2 "This is argument 3")
```

The three arguments can then be extracted by the UserAgent code via a simple call to the Agent method `getArguments()` that returns an array of type `java.lang.Object`.

```
public void setup() {
    ...
    Object[] args = getArguments();
    String arg1 = args[0].toString();
    // this returns String "1"
    String arg2 = args[1].toString();
    // this returns String "arg2"
    String arg3 = args[2].toString();
    // this returns String "This is argument 3"
    ...
}
```

The same arguments can be passed via the in-process interface (hence by extension web scripting through JSP) as follows. ProxyAgent is merely a handler for data validation and transit.

```
Object args = new Object[3];
args[0] = "1";
args[1] = "arg2";
args[3] = "This is argument 3";
AgentController dummy = ac.createNewAgent("user1", "UserAgent", args);
```

Therefore, the CD-ROM does not implement ProxyAgent, but takes command line arguments to our UserAgent to simulate data entry for searching and sealing deals.

By utilising this useful ability to pass arguments to agents, we can start them with root commands, and create a command dictionary which is found in Section 8.1.

Genghis will not become usable unless a web interface is built, an issue discussed at the beginning of this document. For the purposes of this project, it remains enough to allow easy extensibility for a JSP interface but there is no need to provide one as standard.

The ProxyAgents must follow the same development cycle as for other agents, once a developer decides how they want their interface to look, and what it will actually do.

On the HTTP side of servlets, we make use of standard methods. `init()` will setup the servlet when it is started by Tomcat. `doGet(httpServletRequest request, httpServletResponse response)` handles GET requests. `doPost(request, response)` for POST requests. `destroy()` for when the servlet is destroyed (when Tomcat shuts down).

There is a small (non-compilable) code snippet of JSP.

■ Integrate/Rename `/DISSERTATIONJADE/src/genghis/http/template.jav` ■

Chapter 7

ReGreT and reputation

How the first level of reputation calculation using ReGreT can be integrated into UserAgents. ReGreT is oriented to e-commerce environments, so we take the view that pooling deals are customer transactions. Explanatory notes about data privacy classes appear later in this Section. This Chapter applies the ReGreT paper [16].

Individual reputation The first dimension of the ReGreT system, and the one we focus on. It considers direct interactions with other members of the agent society. We restrict the way the information is requested so that it becomes comparable.

Social reputation The second dimension of ReGreT which allows for information coming from other members of the society in terms of *witness*, *neighbourhood* and *system* reputation. This dimension is not covered.

Ontological dimension We will not be detailing this level.

Data Privacy Simple data privacy model applied to the Genghis schema.

7.1 The Problems of Calculating Reputation

ReGreT shows how social analysis can be used to minimise these problems and how it can be incorporated as a new source of information to calculate reputations. In societies, it is possible to identify different types of social relations between their members.

An estimate of reputation is necessarily formed and updated along time with the help of different sources of information. Direct interactions and information coming from witnesses are the main sources to build reputation. Although convenient, they are not

free of problems. Direct interactions are not always available and witness information can be false, biased or incomplete and suffer from the *correlated evidence problem* (CEP).

The correlated evidence problem occurs in the social dimension, within a component called *witness reputation*. In an ideal world, information passed between homogeneous and trusted agents, this information is as relevant as the information from direct interaction (individual dimension). The two things that can happen are - false information is relayed and agents hide information.

An extension to ReGreT proposes a method of lying as a stochastic process to implicitly reconstruct witness observations in order to alleviate this problem. The basic steps to minimise the effect of the CEP are to identify the agents who are witnesses, aggregate the trusted agents, and make questionnaires based on an index of trust reputation.

7.2 Implementing a model for Genghis

We need to create a usable model that is a vital part of the Schema. From the study of ReGreT to an individual, we need to capture certain information.

To use network analysis, we must create a sociogram for each social relation, and the simplicity of agent societies makes this better. However, we need to collect data, which is where the real difficulty lies. The decision to make sure users give as much data as possible was made real by introducing the *feedback pending* process and a separate Table in the Schema of A.3. Always remember that sociograms are dynamic and agent dependent.

We must appreciate that it will take time to create the formulas and implementation details, and this attempt only scratches the surface. The example used in [16] is a supply chain with sellers specialised in a single type of product and can be independent or part of a company that groups several sellers specialised in different products. The buyers buy the products they will sell in another layer of the supply chain. Further details of this good example are in [16].

One thing is clear in that our model will work between UserAgents. Furthermore, we specify a directed relation of type *Cooperation(coop)*, from 3 choices that include *Competition(comp)* and *Trade(trd)* since Genghis agents are neither competitive (vying for the same resources) nor trade pools. They co-operate, and exchange their own information with other agents, trying to help each other as their only effective goal.

We have already said that each agent owns a sociogram, which is for the Cooperation(coop) relation type, and this is a non-directed graph with weighted edges. Weights go from 0 to 1 and reflect the intensity of the relation. Hence we arrive at our *grounding relations* that appear in contracts between Drivers and Participants (Driver or Passen-

ger or Ambivalent) in a sealed pooling deal. A grounding relation is a tuple *(variable,{ +,- }, outcome)*, showing how an increment of the value affects the reputation. There is two variables which are complex types and not grounding relations (quality_swindler and usage_swindler).

On the grounding relation: A + means that if the value of the issue increases, the reputation also increases, while a - means that if the value of the issue increases, the reputation decreases.

For the Driver, we consider these reputation variables.

to_overcharge *[Price,+,0.5].* Tends to return high costs to queries for pooling. This will not happen since agents use the same algorithm to return costs to a CFP. In future, it could happen that several Drivers might be in the position to specify some kind of exclusivity.

to_deliver_late *[Arrival,-,0.2].* Tends to arrive late from their pool appointment time. We will have to define what late really means.

to_notshowup *[DriverDefaulter,-,1].* Tends to not arrive at all to pick up the participants (a rather serious deficiency).

quality_swindler Tends to not really match up to the characteristics specified to the other party. For example, a share offered for 4 people may be in a vehicle that makes it a very tight squeeze for 4 people. This is a complex type and is calculated from a proportion of to_deliver_late, to_notshowup and is_functional.

is_functional *[Functional,+,0.5].* Tends to be generally functional as a pool Driver, and not overly lacking in any respect.

For the Participants, we consider these variables.

to_deliver_late *[ParticipantArrival,-,0.3].* Tends to arrive late for their pool appointment.

to_notshowup *[ParticipantDefaulter,-,1].* The tendency to not show up to their booked appointment. Again, a serious deficiency, but a straight 0 score is not *necessarily* made.

usage_swindler Tends to *use* a share for some other purpose *without making it clear*. If it was made clear that I will have lots of shopping bags, and this was fine with the Driver, then this score should not be affected. This is a complex type calculated from a proportion of to_deliver_late, to_notshowup and is_functional. It is hoped for the future, when Genghis ReGreT implements the social dimension, this variable will be modified.

Field	Contents
Index	A simple primary key
FromAgent	A pointer to the source agent
AboutAgent	A pointer to the target agent
Price	[0-1]
Arrival	[0-1]
DriverDefaulter	[0-1]
Functional	[0-1]

Table 7.1: Driver Outcomes Database Scheme

is_functional *[FunctionalParticipant,+,0.5]*. Tends to be generally functional as a pool participant, and not overly lacking in any respect in their role as a participant. The dishonesty about intentions (such as the shopping example) can allow a reviewer to give a score of 0 here.

The way to use ReGreT is to take relations and calculate *reputation* giving weight to more recent outcomes, into an *outcomes database*. We will not implement code to weight recent outcomes though. Our implementation will store values associated to each agent as shown in the example Driver Table 7.1, which have also been incorporated into the Schema in Figure A.1. The Participant Table is quite similar and so it is not given here. Complex types are not stored because they are calculated on the fly.

Remember that the value integer stored for each variable is not the third component of the grounding relation, but the weight of the issue. The third component of our relation tuple specifies an *importance* weighting to the issue when data is aggregrated.

The command dictionary has a UserAgent command `querymyreputation` which starts the `QueryMyReputationBehaviour` instance. We generate sample data via the TestingAgent and its `GenerateReputationDataSQL` behaviour. There are 50 records for Participants in random journeys, logged at the end of the seeding file (Section 8.1.4).

■ `/DISSERTATIONJADE/createjourneysdb.sql` ■

We can add to a relevant outcomes database by commanding `addmydriverreputation` and `addmyparticipantreputation`, as detailed in the command dictionary of Table 8.1.

7.3 Data Privacy Classes for the Schema

Data privacy does not refer to the accessibility on the data, but refers to the availability of data between agents. Data is tagged to five states where the Private and Public states

may not be transitioned to another state.

Private Denoted **P**. Strictly private data in any process. No agent may divulge this data except ProxyAgents for transit and authentication.

Public Denoted **A**. Public data in all processes. Agents may communicate this information within Genghis.

Contingent Contigent protection maps private to public data in a state such as one where a carpool deal has been sealed, and certain data which are normally private need to be exchanged by the two parties. Contingent protection is available at two levels.

  Search Only Denoted **C-SL**. Data fields available during queries only.

  Post Confirmation Denoted **C-PC**. Refers to the changed property of an SL field after a deal is sealed, and makes this data field available post confirmation only.

Group View Denoted **C-G**. In an alternatepool group, certain parts are visible only to the group, and are private outside the group.

The above references have been appended into the schema of Appendix A.3 and are used as a guideline for coding. Where the labels show Contingent states, further explanation is in the Notes which follow (Section A.4).

We maintain that the enforcement of data privacy has to be done at schema level and not merely at the ontology level, since many ontology content slots are derived from the database.

Further to the above, we appreciate that multiple installations of Genghis (if they ever work as an extranet) demand new thoughts into further categories of privacy. Indeed, it was with this intention that the Journeys Table was designed such that it would be the *main* entity that was shared between other installations of Genghis in future. It is hoped that all Genghis installations maintain their common ontology, which would allow a data exchange standard to be made concrete. This is explored in Chapter 10.

Chapter 8

Testing and Seeding

This Chapter examines test methodologies and results. Seed data are devised and loaded.

Sample Data We make extensive use of a temporary agent called TestingAgent with behaviours that can be switched on/off. This agent generates SQL statements with random data which is captured from the terminal and loaded in mySQL.

Testing Requirements After loading data, we write a command dictionary to test agents with input.

8.1 Seed Data

8.1.1 TestingAgent

We use a specific agent to generate seed data. It has behaviours to switch on and off depending on the dataset being created. The primary keys for the data can also be controlled by modifying the loop counters.

Each behaviour in this agent is capable of a unit test. For example, a basic database interaction engine is able to test samples of code that parse output from the database.

■ `/DISSERTATIONJADE/src/genghis/TestingAgent.java` ■

8.1.2 Users

30 seed users are entered into the Users Table. There is 10 Drivers, 15 Passengers and 5 Ambivalents. We use them as a basis for modelling their journeys and reputation.

8.1.3 Journeys

SQL is derived for a randomly generated set of 80 Journeys entered into the Journeys Table. We use GenerateJourneySQLBehaviour in the TestingAgent to deliver a terminal output to capture and load. The relevant supporting methods are given in the source code of TestingAgent.

The saved capture is logged into the main creation script.

■ `/DISSERTATIONJADE/createmysqlschema.sql` ■

8.1.4 Journey-User Associations

We once again use a TestingAgent behaviour to calculate random associations between users and journeys to place into the JourneysDB Table. We want to ensure a fair distribution of these between the users. The use of pseudo-random generators in Java is sufficient.

Custom tweaking of the GenerateJourneyDBSQLBehaviour lets us create different association for three roles of Driver, Passenger and Ambivalent. The SQL which TestingAgent prints is filed separately.

■ `/DISSERTATIONJADE/createjourneysdb.sql` ■

Additionally, there is a behaviour that generates data for reputations randomly for random participants, and there are 50 such records for Participants generated by the TestingAgent behaviour GenerateReputationDataSQL - logged into the same seeding file.

8.2 Command Dictionary

As identified in Section 6.2, we do not need a GUI to test Genghis, and in fact a GUI would be cumbersome. The agents take arguments from the command line and we construct a basic form for these commands as given below. Note that this loads agents into the main container, when ideally one should use child containers and keep the main container active.

```
java jade.Boot userN:UserAgent(command arg1 arg2 ...)
```

where `command` is a string command in the dictionary followed by the arguments in the correct order for that command. As usual, we are careful with naming the agent `user`

followed by the primary key N of the represented user in the database. We can retrieve values in the `setup()` method of agents.

```
Object[] args = getArguments();
String command = args[0].toString();
if (command.equals("mycommand")) { ... }
```

Table 8.1 is the table of commands. The String types we retrieve must be parsed into the correct types.

Agent	**Command (first argument)**	**Ordered Argument(s) [DATA]**
UserAgent	startcfp	ToLONG[float], FromLAT[float], JState[Wanted], Quota[int], FromLONG[float], JType[Standard], Duration[int], ToLAT[float], StartTime[HH:MM:SS], Date[YYYY-MM-DD]
UserAgent	sealthedeal	UserID[int], JRole[1 or 2], JourneyID[int], Quota[int]
UserAgent	addactivejourney	UserID[int], ToLONG[float], FromLAT[float], JState[1], Quota[int], FromLONG[float], JType[0], Duration[int], ToLAT[float], StartTime[HH:MM:SS], Date[YYYY-MM-DD], JRole[0]
UserAgent	addwantedjourney	UserID[int], ToLONG[float], FromLAT[float], JState[0], Quota[int], FromLONG[float], JType[0], Duration[int], ToLAT[float], StartTime[HH:MM:SS], Date[YYYY-MM-DD], JRole[1 or 2]
UserAgent	querymyreputation	*none*
UserAgent	addmydriverreputation	FromAgent[int], AboutAgent[int], Price[int], Arrival[int], DriverDefaulter[int], Functional[int]
UserAgent	addmyparticipantreputation	FromAgent[int], AboutAgent[int], ParticipantArrival[int], ParticipantDefaulter[int], FunctionalParticipant[int]

Table 8.1: Agent command dictionary to be used for launching agents

An issue that arose with adding active and wanted journeys is that they happened in two stages - first the journey is added to the Journeys Table, then an associated entry

is made in the JourneysDB table. However, to make the associated entry, we use the LAST_INSERT_ID() mySQL command, which means that although very rare, it remains possible during very heavy use to get an incorrect *last primary key*.

```
/DISSERTATIONJADE/src/genghis/db/DB.java:
public String getLastInsertedID() throws SQLException{
String query = "SELECT LAST_INSERT_ID() AS id";
ResultSet rs = executeQuery(query); ... }
```

8.3 Simple Testing

This Section contains simple scenarios we can make with our sample data and command dictionary.

Start the following system with one of the agents asking for CFPs. You could boot in a container if desired, but deregister agents with the same name first before launching them, since each agent will automatically try to register itself with the DF. Notice that UserAgents must always contain at least one argument, for example none.

```
java jade.Boot -gui user1:src.genghis.UserAgent(none)
 user2:src.genghis.UserAgent(none)
 user3:src.genghis.UserAgent(startcfp 4.9876 3.456 Wanted 1 4.7654
 Standard 30 2.4565 14:15:00 2004-04-21)
```

Test input is not validated, so they should be the correct data in the correct order.

Many more command examples are written on the README file.

■ /DISSERTATIONJADE/src/genghis/README ■

Chapter 9

Extensions

Ideas noted in point form, on practical or conceptual extensions to make Genghis powerful and user friendly.

User extensions Making the system show complex understanding of trustworthiness and increasing the accuracy of calculating reputation through the next level of ReGreT. Graphical and usability guidelines for developers writing GUI layers.

Route matching extensions Allowing better visualisation of a route and providing the means to select shortest or quickest route into the cost returned from a CFP. Building further tolerances specified from the user side, beyond specifying time importance.

MAS extensions Clustering computation for intercity or international carpooling - the concept of making start and end destinations *fuzzy*. Creating institution-level location nodes with a GIS, but having to solve the classical NP-complete problems of Vertex Covering and q-Clique to work out which nodes are institutions by looking at connectivity to other nodes. The concept of a Node itself is to be changed in that a small region, and not a point, should be called a Node.

Temporal Connectives Further research into a logic of temporal grants and rejects and their technical relevance to Jade performance. Do they give a little more power to make search/seal-the-deal express contingency, or are they not necessary?

Alternatepool lifecycles give us Eventpool? Finding a mapping between the needs of the non-primitive eventpool type and an alternatepool with a time-to-live extension (ttl). Modelling what happens at a very high intensity/throughput node where an event generates a hugh turnover of searches/deals at some nodes.

Interaction extensions Why SMS and email command modules would be useful to drivers and passengers taking journeys. The risk of misuse of each extra channel of interaction. RDF publishing of journey databases for indexing at a central Genghis directory, or a Global Genghis Directory Service, making the discovery of datasets standardised. Genghis interconnectivity is the subject of the entire Chapter that follows this one. Phone based interaction by voice command with text to speech conversion and vice versa - allowing commercial applications to generate revenue and spread Genghis application usage.

Handheld/Mobile Extensions Facilitating match pairs with guidance on optimal routes to a destination through existing in-car navigation systems. Allowing GPS/navigation equipment to autoadvise a local Genghis mirror about an impending Active journey. The easier we make it for a Driver to add their Active journeys, the more useful potential sharers will find it. A JINI car sharing (not trip sharing) agent implementation is [15]. This area should focus into the use of *aglets* which are mobile Java agents. Naturally, issues in Human Computer Interaction (HCI) will arise.

Genghis beyond carpooling Specific Genghis agents with specific reputation bounds can bid for mail courier/goods transport. Integration with other transportation MAS. Having proposed this however, there is other hurdles to overcome. [7] contains a proof that the Routing Decision Problem (RTD) is NP-hard and subsequently shows that it is a polynomial reduction of the Rural Postman Problem, and therefore NP-complete.

Execution Dynamics Testing strategies to build cost weighting with traffic jam simulation or real-time congestion feeds for user agents. Stochastic traffic jam generators are available, as a reading of [7] will show. Can the DecisionModule factor traffic jams into a singular costed return? The possibility of feeding real UK traffic information [18] would be groundbreaking. Integrating open Genghis clients into new vehicles can then work pervasively as a highly evolved open source project, and gain the attention of car makers like Mercedes-Benz and Ford to have interoperable Genghis clients built into a concept car that knows about traffic avoidance.

Introducing Dependent Users The concept of a user with a UserID can be extended to include dependent users. The idea here came from the way a school pool works. In alternatepools, if we assume that the users are parents dropping off their children to school, we wish to separate the alternatepool of their children and interact with the parent users within the alternatepool, since the number of children per parent will determine how many credits the parents have pooled in total. It is not practical to make each child have their own Genghis user account.

Bad extensions There may be ideas that devolve the spirit of Genghis and remain unsuitable as extensions, despite the attraction. One of them could be the use of *fixed node names in text* **or** strict co-ordinate decimal accuracy to identify places. By doing this, we have no edge over a standard database-driven web application, and this is why we have leaned toward latitude and longitude. In the future, latitude/longitude references might be *clustered* or become *fuzzy*, but it remains an open option to do this. One can search for a share from London to Birmingham, where the entire area bounded by these two cities is converted into a large but temporary *single* node for runtime.

■

Using latitude/longitude makes intercity/international travel as workable as a 2 mile trip to the village shop.

■

Chapter 10

Distributed Genghis

The nature of a distributed set of applications, connection interfaces and host agents. Sharing local data via interfaces to other installations of Genghis has privacy and usage problems.

Directory Service The need for a global directory to with which all local installations can register. The more important need to build common ways of messaging platforms across the global directory.

Protocols and Open Upgrades It is very likely that local installations might modify or improve parts of Genghis. We need to deal with the upgrades which should be common and upgrades which are optional, and Global Genghis plays a role in distributing these.

10.1 Global Genghis

We call the set of all interconnected Genghis installations the Genghis extranet, which uses a service called Global Genghis as a DF and protocol library.

Agent mobility is a feature where agents are location-aware through `jade.domain.mobility`. The reason we might not want to use a large interconnected set of Jade containers (as Figure 10.1 shows) is because developers may want to extend local features into their installation. Further, an offline or bogus installation could easily destroy the integrity of data and affect people, discrediting the application.

There is clearly a need for a single entity that all Genghis installations will commonly know as a directory service to other Genghis installations (like a web DNS server). This is a project that stands on its own as well as integrates MTP designs to every

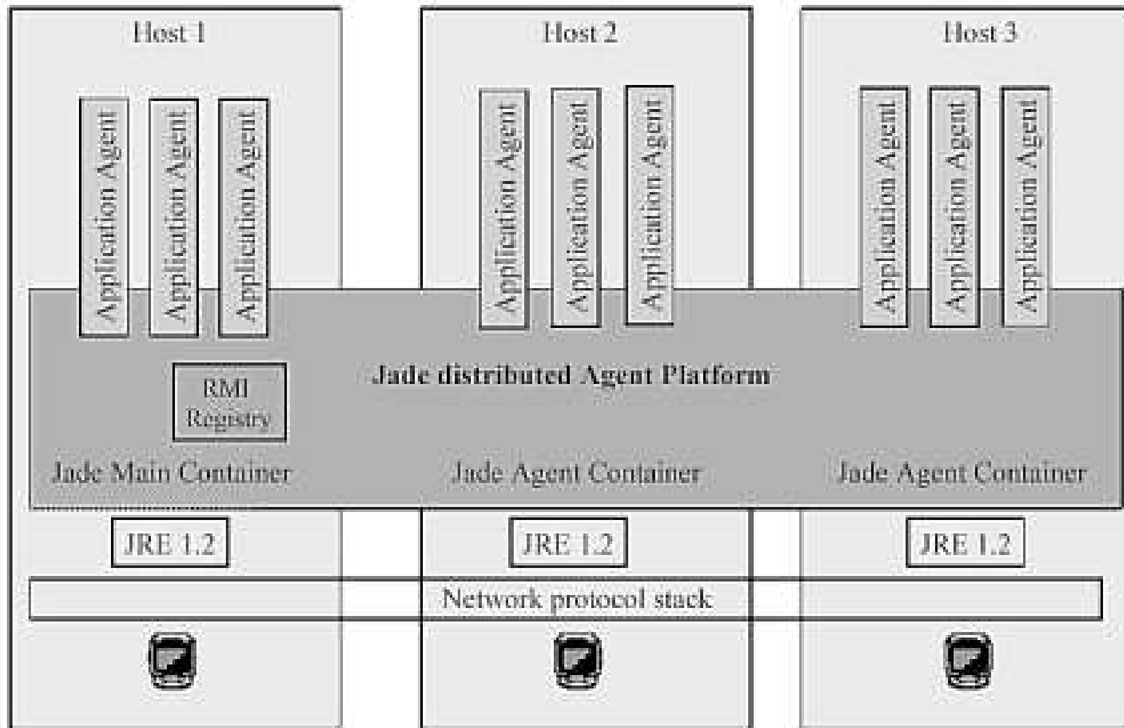

Figure 10.1: Jade Agent Platform across containers (Reproduced from Jade Programmers Guide)

local installation of Genghis, which has to register itself with Global Genghis upon bootstrapping the local system.

■ Package `jade.mtp` contains a Java interface that every Message Transport Protocol (MTP) should implement in order to be readily integrated into Jade. ■

Without a central DF, agents cannot locate journey datasets and service users for good journeys outside their own installation.

The nature of this project demands data sharing (with inherent additions to our privacy model) and distributed processes of querying and returning results. Fortunately, Jade allows agents to clone and migrate across containers, with certain restrictions. We do not face many issues during the search for a journey across containers, however the current notion of *seal the deal* has to completely overhauled for the following reasons.

External bookings A deal which is sealed by a journey being driven by a user on the extranet has to be labelled as an external deal for formality. The natural issue arises about data privacy remaining the same across extranet deals, since

the foreign installation might want to impose more/less strict access to its data to the querying installation.

Reputation migration After highlighting extranet deal scenarios, we must make a decision about whether all reputation data is first - common across the extranet and second - universally accessible across the extranet. It is most likely that all reputation data will be common and universal since private data are never revealed about a user.

Standard Issues in Distributed Networking System downtime and local change may not be communicated properly to the interested parties on the extranet. For example (in our current system), a local change to a journey cannot be applied after a deal is sealed. This is a good thing since that change may not be acceptable to the journey participant (extranet or local). A journey must be cancelled if there is any change and recreated with new parameters, which lets extranet participants trust a deal in the knowledge that it will not be changed.

Single Data Instances We should address the preservation of journey data integrity. In effect, we must decide whether journey data is ever mirrored (to work around a local system being down) and how failsafes would work after an extranet deal is sealed.

10.2 Versions and Upgrades

It is very likely that better DecisionModule code or plugins/upgrades may be made available for Genghis, especially for multihop journeys, and the various extension ideas that we explored in Chapter 9. By following examples of large scale open source (GNU GPL) projects, we must find a way to automatically distribute the *critical/common* upgrades and list *optional* upgrades.

Two ideal facilitators are - a web portal for optional upgrades and Global Genghis for the common upgrades. Registration by a GlobalGenghisRegistrationAgent for example, can concurrently find out new common upgrades and patch the local installation.

Appendix A

MAS and Database Schema

A log of Gaia tables constructed to define all Genghis agents according to this methodology. We discuss details and follow up with design. The second part deals with the database schema to support this MAS. We have considered basic requirement notes in the body prior to writing agent services.

A.1 Details

[13] provides a walkthrough of MAS development for Jade using Gaia, from which the following is summarised.

Firstly, we should remember that there is no given way to go from a Gaia model to a system design model. Gaia defines the structure of a MAS in terms of a role model. The model identifies the roles that agents have to play within the MAS and the interaction protocols between the roles. The objective of the Gaia analysis process is the identification of roles and the modelling of interactions between roles. Roles consist of

Abstract	**Concrete**
Roles	Agent types
Permissions	Services
Responsibilities	Acquaintances
Protocols	
Activities	
Liveness Properties	
Safety Properties	

Table A.1: Abstract and Concrete concepts in the Gaia methodology

Operator	Interpretation
x.y	x followed by y
x OR y	x or y occurs
x^*	x occurs 0 or more times
x+	x occurs 1 or more times
x^π	x occurs infinitely often
[x]	x is optional
x INT y	x and y interleaved

Table A.2: Gaia operators for liveness formulas

four attributes.

Responsibilities This is the key attribute of a role, as they determine functionality. Responsibilities are of two types.

- **Liveness properties** The role has to add something good to the system, in other words - tasks to fulfil. We use a set of operators to form liveness-expression formulae, as given in Table A.2.
- **Safety properties** The role must prevent and disallow that something bad happens to the system (ensure acceptable state of affairs during execution cycle).

Permissions Represents what the role is allowed to do and what information resources it is allowed to access.

Activities Tasks that an agent performs without interacting with other agents.

Protocols The specific pattern of interaction e.g. a seller role can support different auction protocols.

The Gaia process has the following steps.

The first step is to map roles to agent types. Agent types can be an aggregation of one or more agent roles. The second step is to determine the services model needed to fulfil a role in one or several agents. A service is derived from the list of protocols, activities, responsibilities and liveness properties of a role. Finally, the last step is to create the acquaintance model for the representation of communication between the different agents, which is a simple graph showing connection pathways between agent types.

A.2 Genghis agents

We start with the Gaia role models presented in Tables. The basic requirements are from statements in Section 4.3.

Role UserAgent
Description Service inside the Jade Container, in contrast to ProxyAgent which serves the web side of user processes. Allows a user all the actions that the system can offer. It may prohibit, validate and proactively engage in agent communications with respect to the user. The range of interactions include push, post and list. Each activity engages with no agents, other agents or the database. Outside dynamic influences are the JourneyNotifyAgent who informs it about a wanted journey being fulfilled. The booking is up to the user who is advised once only by email/SMS. Journey acceptance after message receipt from JourneyNotifyAgent is first come first served, hence a ValidateJourneyAcceptance activity. A set of gateway behaviours is a service to and from ProxyAgents (optional).
Protocols and Activities ServeUser, HandleNewEvents
Permissions Read, Edit, Append to database. Interaction with all agents.
Responsibilities
1 Liveness
InitUser.(DBRetrieve) OR $(\text{ServeUser})^{\pi}$ OR $(\text{HandleNewEvents})^{\pi}$
DBRetrieve = ExecuteDBQuery
ServeUser = JourneySearchCFPResponder OR AbstractProxyAgentInteractionService OR SealTheDealBehaviour OR AddActiveJourneyBehaviour OR AddWantedJourney-Behaviour
HandleNewEvents = JourneySearchCFPInitiator OR AddMyDriverReputationFeed-back OR AddMyParticipantReputationFeedback OR QueryMyReputationBehaviour
2 Safety True. Successful connection to database and registration with DF.
Protocols Contract Net Protocol

Table A.3: UserAgent Gaia role model

Role ProxyAgent
Description Serves the web side of user processes, which is the GUI, and does first level data validation. Allows a user all the actions that the system can offer. It may prohibit, validate and proactively engage in agent communications with respect to the user. Each activity engages with other agents but not the database.
Protocols and Activities InteractionHandler
Permissions Interaction with all agents.
Responsibilities
1 Liveness
InteractionHandler = (PushBehaviours)$^{\pi}$
PushBehaviours = UserJourneyQueryPush OR UserConfirmJourneyPost OR UserWantedJourneyPost OR UserReceiveWantedMatch OR UserAddActiveJourneyPost OR UserListAllJourneys OR UserJourneyFeedbackPost OR UserListFeedbackPending
2 Safety True. Successful connection to Jade established.

Table A.4: ProxyAgent Gaia role model

Role JourneyRoundupAgent
Description Constantly monitors the active journeys dataset and labels journeys as complete after records show that they are complete according to their date, time and duration. Post a record into the database for the user to prompt feedback on the completed journey.
Protocols and Activities JourneyCyclicFlagCompleteJourneys, ChangeJState, JourneyFeedbackPendingPost
Permissions Read, edit, append to database.
Responsibilities
1 Liveness Monitor = (DBMonitor)$^{\pi}$.(ChangeJState)
DBMonitor = GetJourneyData
ChangeJState = CheckJCompletion.ChangeJIDJState
2 Safety True. Successful connection to database established.

Table A.5: JourneyRoundupAgent Gaia role model

We list an interactions model following the role models.

Protocol SearchForJourney
Initiator(s) UserAgent, JourneyNotifyAgent
Receiver(s) UserAgent, JourneyNotifyAgent
Responding Action ServeUser, HandleNewEvents
Parameters The central query engine which gets criteria for a journey and goes through

Role JourneyNotifyAgent
Description Continually compares each item in the wanted journey dataset with the entire active journeys dataset. Uses the standard costing algorithm to probe journey relevance and posts the best resulting cost (WIndex) as an update into the journey record. Whichever UserAgents that are interested can retrieve the latest WIndex score of a wanted journey. If a WIndex is above a certain bound, inform UserAgent(s) that are associated to this JourneyID.
Protocols and Activities JourneyCyclicFlagCompleteJourneys, ChangeJState, JourneyFeedbackPendingPost
Permissions Read, edit to database. Interaction with UserAgent.
Responsibilities
1 Liveness Monitor = $(\text{DBMonitor})^{\pi}$.(AssessCostJourneysAlgorithm) DBMonitor = GetJourneyDataW.GetJourneyDataA
AssessCostJourneysAlgorithm = CheckforSingleRecord OR UpdateDBWIndex OR CheckWIndexBound. [InformJMatch.EmailUserID]
2 Safety True. Successful connection to database established.
Protocols Contract Net Protocol

Table A.6: JourneyNotifyAgent Gaia role model

the CNP.

Protocol RespondtoCFP
Initiator(s) UserAgent
Receiver(s) UserAgent, ProxyAgent
Responding Action Depends on whether agent initiated or is simply responding - ServeUser, HandleNewEvents
Parameters Response to a CFP as a proposal or collation of an (initial) initiation.

Protocol SealTheDeal
Initiator(s) UserAgent
Receiver(s) UserAgent, ProxyAgent
Responding Action InteractionHandler, HandleNewEvents
Parameters Confirmation from the web user is necessary to book a deal.

Protocol InformWantedMatch
Initiator(s) JourneyNotifyAgent
Receiver(s) UserAgent
Responding Action AssessCostJourneysAlgorithm
Parameters Send an email for the green bound match news.

Protocol AddWantedJourney
Initiator(s) UserAgent, ProxyAgent
Receiver(s) UserAgent
Responding Action AddWantedJourney
Parameters Adds a new journey that a user wants to keep a lookout for.

Protocol ProcessFeedback
Initiator(s) UserAgent, ProxyAgent
Receiver(s) UserAgent
Responding Action ServeUser, HandleNewEvents
Parameters The JourneyRoundupAgent monitors completed journeys, but it is UserAgent who actually displays pending feedback and handles it when it does come in.

For the Gaia design phase, it can be seen that our agent system is simple and hence we shall not draw the interconnection graph to show which agent is related to which. This is already implicit.

Our Gaia Services/Acquaintances Model is a reworded version of what we have seen thus far, and Section 2.5.2, where the pre/post conditions are implied through the role model. Again, our small number of agents and interactions model makes it clear in that we can discern inputs and outputs. The output from some processes is a confirmation to a web user that an action has been carried out. Gaia simply connects acquaintances, which is something we have already implied through the specific interaction roles of each agent.

It should be noted that implementing the liveness expressions is self-evident in Jade, as the developer can introduce Cyclic, Sequential or Concurrent Behaviours in our agent thread.

The Javadoc HTML for the codebase is given.

■ `/DISSERTATIONJADE/REFERENCE/src/index.html` ■

A.3 Database schema

Figure A.1 shows a schema derived from parts of the ontology. The intention is that agents will interact directly with the database, which is implemented in mySQL. The schema has been normalised to third normal form and the notes following the schema should be studied. Data privacy levels (P, A, C-SL, C-PC, C-G) are described in Section 7.3. Primary and foreign keys have no privacy status since they are indices.

It would be better to abstract database interaction to an agent, so that various DBMS can be used with Genghis, but this can built in the future. We can optionally use the highest performance open-DBMS currently available - *InnoDB* with mySQL 4.0. We shall not configure it for Genghis, but migrating is straightforward.

■ mySQL (DBMS) script at `/DISSERTATIONJADE/createmysqlschema.sql` ■

A.4 Schema Commentary

The items in this Section are organised by Table or Issue. Figure A.1 shows the schema tables and should be studied before reading this Section. The schema shows required fields in bold type with PK standing for *Primary Key* and FK standing for *Foreign Key*.

JTypes This table is a holder of the pool type applicable to a journey - i.e. Standard or Alternatepool, definitions of which are in Chapter 3.2.

JRoles This table is a holder of the three roles of Driver, Passenger and Ambivalent.

JState This table is a holder of three journey states - Wanted, Active or Complete. The complete state in a journey cannot be transitioned into by a human user or any other agent except the JourneyRoundupAgent which makes it complete when the end time has passed. We appreciate that this state will be Active *during* a journey being driven as well as before the journey. This is fine since we use Journeys.StartTime to verify that passengers have a certain amount of time (for example a minimum of 30 minutes) before a journey booking is entered.

AlternatepoolGroups A UserID becomes a member of a group in two ways - by directly applying and having a record pending moderation in GroupsDB or by finding an Alternatepool journey that they want - which demands that they get moderated into the group. In both cases new group members are pending moderation in GroupsDB. UserID in this table is the user who founded this group. New groups may be founded by any UserID, and if no other UserID is in that GroupID in GroupsDB, the record is set to GroupsDB.Pending = false and AlternatePool.UserID = UserID since there is nobody to moderate someone into a newly formed group. Hence UserIDs can create their own groups or join existing ones. Group privacy is taken care of since founders decide who can join through moderation.

Users Attributes describing the user entity. The RoleID saved here is requested during some processes to make sure users do not masquerade as Drivers when they are only recorded as Passengers and so on. The UserID will be the same as the agent

AID, so it is important to seed Genghis on the bootup process with the range of integer values from 0 to the number of records in Users (since UserID is the primary key). The decision to reveal an email address (privacy code C-PC) is a difficult one. To keep things open, we let the email be seen only after any current/future deal (for any active journey) has been sealed by *the user trying to access* the normally private email address.

Journeys This table is necessarily a separate entity (except some foreign keys) since we may wish to share journey datasets in future. If JType is an alternatepool, the act of adding this journey into JourneysDB will create a record pending membership moderation in GroupsDB. WIndex is the field where the latest traffic light score is stored in case the journey instance is of JState `Wanted`. Also, this field is not accessed for active journeys, which is why the privacy is set to contingent mode for search only. We do data validation and bound checks on Quota, Date and StartTime prior to adding a record.

JourneysDB The central table where journey relationships are managed. The UserID refers to the user wanting or adding the journey. The JourneyID concerns a given journey and has constraints as shown in the note following this one. RoleID is a given UserID role in this journey. This item is cross-checked with Users.RoleID to make sure the user is capable of being in this role such that he does not masquerade as a Driver when his Users record shows that he is only a Passenger. The GroupID is an optional field allowing this journey record to be associated with a group. An entire entry row is disallowed if UserID is pending membership for a supplied GroupID or is not a member of the given GroupID at all.

JourneysDB - JourneyID constraints If the JType is `Standard`, then GroupID is null, since standard pools do not involve any groups. If the JType is `Alternatepool`, then GroupID is required. Furthermore, the JourneyID JState can be `Wanted` or `Active`. In both cases group membership is checked and the record is either added directly or added with a related new record in GroupsDB pending moderation. See Figure A.2.

FeedbackPending Entries into this table are only made by the JourneyRoundupAgent. This may expand in future to note down which UserID wrote the feedback about the target UserID. At present the UserID field refers to the user receiving the feedback, and there is no field to save the user who supplied it.

GroupsDB A Table of all memberships to all groups that a user might have for alternatepools. The Pending field is a boolean which refers to the membership of a given user still pending for a given group. If that user has not been moderated into the group yet by the founder it shows `false`. Credits maintains a current signed

integer on the credit this user has for their group membership. We may wish to expand this in future to also say which user has provided/taken credit away from a given user. It is likely we will have do this since a user would be overused or underused if we do not record inter-user credits. Data fields in groups are private to members of the group.

DriverReputation and ParticipantReputation These Tables are a draft resolution of our basic implementation of ReGreT in Chapter 7.

Normalization A relation is in *first normal form* if all columns are single-valued - which is certainly the case in our data model. *Second normal form* is violated when a non-key relation is dependent on a component of the primary key. There is no composite keys that determine any field in the tables. The foreign keys (FK) that point to primary keys (PK) in other tables are pre-loaded with information. The table requiring decisions on association is JourneysDB. Between UserID:JourneyID:RoleID:GroupID there is a many:many:many:many relation. In FeedbackPending between UserID:JourneyID there is a many:many relation. Considering permutations of each column carefully should show that this is fine, according to the notes on JourneysDB above. Before adding rows to the higher level/foreign key tables we must ensure checking for a row with the same data and creating a duplicate, since there is no protection for this through primary keys. A relation is in *third normal form* if it has no transitive dependencies. Every field must be fully functionally dependent on the primary key of each Table. In simple terms, we should be able to add a row to a given table without letting any field be independent of the primary key. It can seen that this is the case for our models.

Journey Recurrence If a journey recurs at periodic intervals, until a certain end date, the record is duplicated in Journeys as if these were separate journeys. We wish to avoid the complications of dealing with recurrence expressions at this time.

Field types The individual data types in fields range between integers and floats to larger sets of alphanumeric characters. We have to use the ones supported by the DBMS chosen which is mysql and details are in the SQL creation script. Latitudes and Longitudes in the Journeys table are signed float numbers, whose characteristics are in Section 6.1.

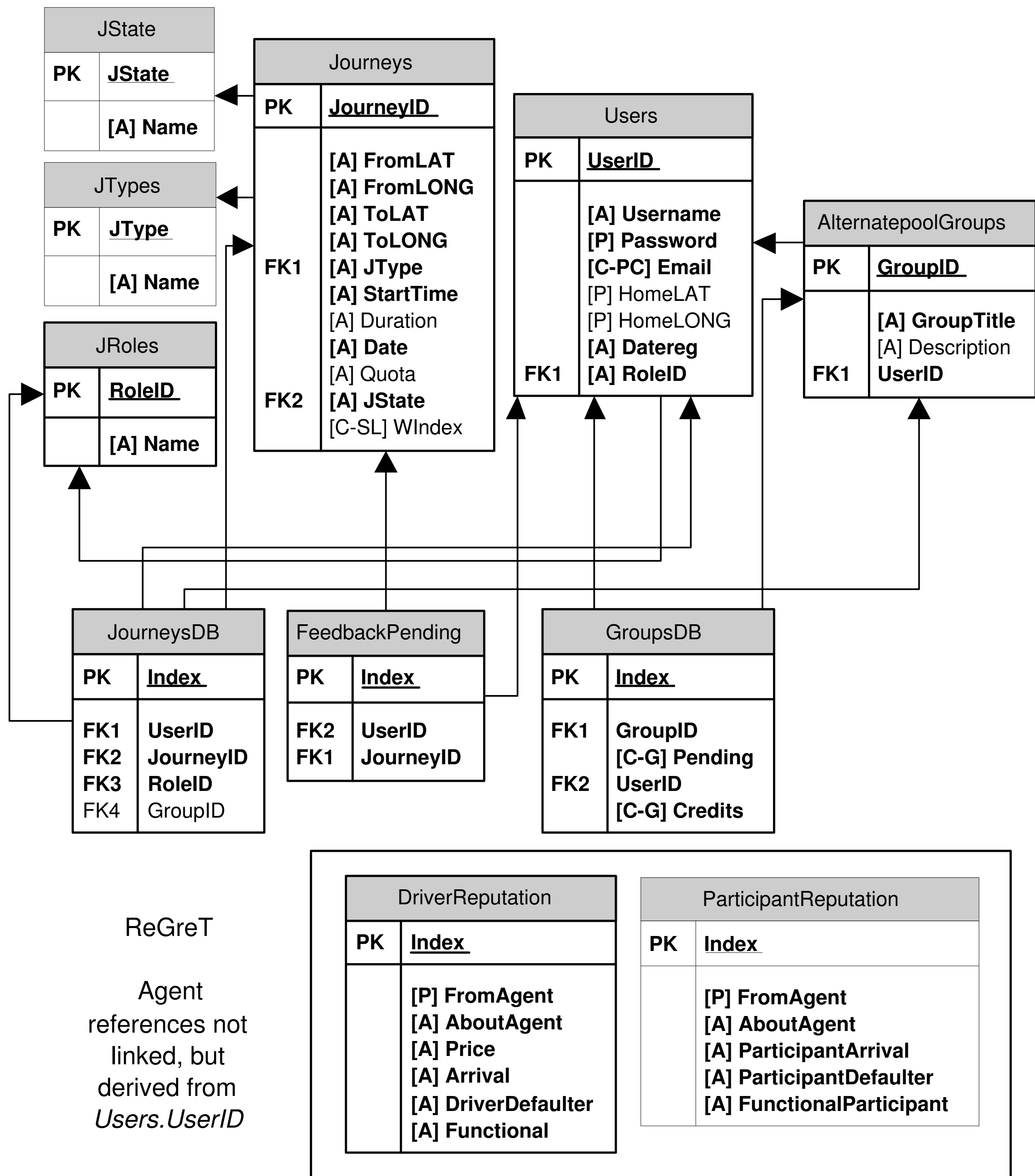

Figure A.1: Schema for Genghis with Privacy codes within an installation

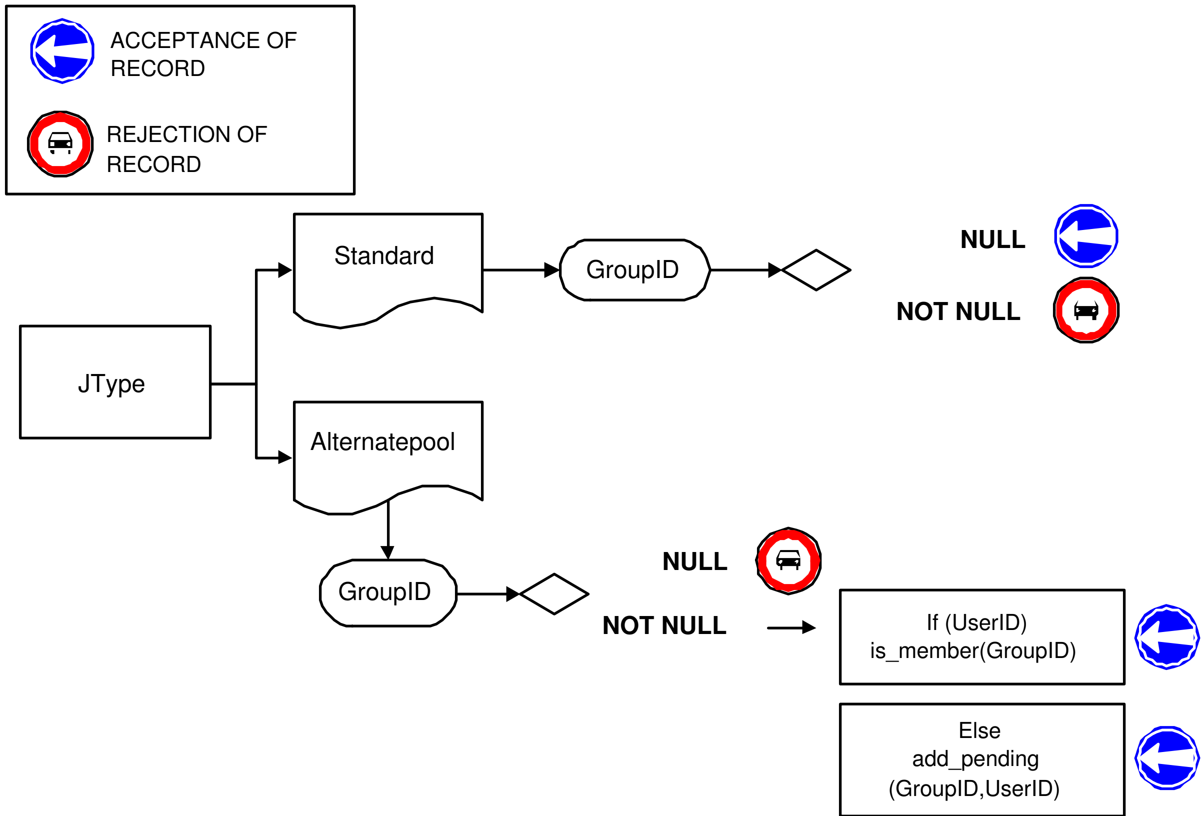

Figure A.2: JourneysDB constraints for JourneyID field

Appendix B

Usage Guide

Here we provide guidelines to set up Genghis, which cover instructions on starting and using the system. It is advisable to look at the README file as well as this Appendix. To understand the agent behaviours, it is necessary to look at detailed comments in the code.

B.1 Pre-Requisites

The following were correctly installed.

■ Platform information at `http://agentcities.cs.bath.ac.uk` ■

JDK 1.4.2_03 Java JDK available from Sun.

JADE 3.01b Available from `http://jade.cselt.it`

JDBC MM.MySQL Connector/J 3.0 A JDBC driver provided by mySQL.

mySQL 4.0.18-nt This implementation uses the mySQL DBMS. However, this can be changed by configuring another driver. We chose not to use an XML Database since portability can be achieved by exporting mySQL data to XML.

If developers were to extend top layer GUI code, Jade is the underlying agent system with which servlets interact. Servlets are processed by Tomcat which is configured for Apache using the module mod_jk so that Apache can communicate with Tomcat. JSP uses the Tomcat JASPER page compiler to generate dynamic web pages embedded with standard HTML and Java directives.

B.2 Configuring and Compiling

You must add the JDBC connector located on the CD-ROM to your CLASSPATH.

```
/DISSERTATIONJADE/src/genghis/db/mysql-connector-java-3.0.11-stable-bin.jar
```

Copy the entire directory tree from node `/DISSERTATIONJADE/` to wherever you wish to store the source.

From this point on and through to compilation, refer to the README file.

■ `/DISSERTATIONJADE/src/genghis/README` ■

After a correct installation of mySQL, create a database called ghengis, and load the following SQL.

```
/DISSERTATIONJADE/createmysqlschema.sql
/DISSERTATIONJADE/createjourneysdb.sql
```

The Java package is rooted at **`src.genghis`**. The parameters to be changed are the database connection information.

```
/DISSERTATIONJADE/src/db/ConnectionDB.java
```

B.3 Running

We are now ready to launch in a UNIX-like operating system. Either a script or the commands in the README file can be used. It is **very important** that UserAgents are named userX where X is the full range of primary keys in the database table Users. Agents will automatically register with the DF.

Depending on what we want to do, we have to issue certain commands. All these are to be found in the README file and the command dictionary can be used for reference to interact in real time with Genghis.

In an X-Window environment, the Jade AMS GUI switch (-gui) can be added to show the AMS and DF interfaces.

The UserAgents which have been pre-programmed with sample data should be launched.

A better way would be to write yourself a script that finds the range of primary keys from the mySQL Table Users, and then generates a command to launch all the representative

UserAgents. Adding and removing users dynamically is possible if each agent was in its own container and uniquely named in context of all containers.

It is important that agent names we give at launch are the same as the primary keys in the database, and no more or less.

■ Name your UserAgents `userN` where N is each and every one PK. ■

For commands with which you can start a system, please refer to Section 8.3 and the README.

Bibliography

[1] Voisard A. Mapgets: A tool for visualizing and querying geographic information. *Journal of Visual Languages and Computing*, 6(4):367–384, 1995.

[2] Matylis G. Burmeister B., Haddadi A. Applications of multi agent systems in traffic and transportation. *IEE Transactions on Software Engineering*, 144(1):51–60, February 1997.

[3] Hildmann H. Carbonaro A., Maniezzo V. An ants heuristic for the long-term car pooling problem. *Chapter 6 in New Optimization Techniques in Engineering*, Godfrey C. Onwubolu and B. B. Babu Eds., 2002.

[4] D. Cliff. Evolution of market mechanism through a continuous space of auction-types. Presented at ABA02, Las Vegas. HP Labs Technical Report HPL-2002-128, 2001.

[5] Meyers D. Dailey D.J., Loseff D. Seattle smart traveler: Dynamic ridematching on the world wide web. *Transportation Research C*, 7:17–32, 1999.

[6] Kephart J. Tesauro G. Das R., Hanson J. Agent-human interaction in the continuos double auction. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-01)*, 2001.

[7] Pischel M. Fischer K., Müller J.P. Cooperative transportation scheduling - an application domain for dai. Technical Report RR-95-01, Deutsches Forschungszentrum für Künstliche Intelligenz GmbH, Erwin-Schrödinger Strasse, Postfach 2080, 67608 Kaiserslautern, Germany, 1995.

[8] Blumenthal C. Michalak S. Goble B. Garner M. Haselkorn M., Spyridakis J. Bellevue smart traveler: Design, demonstration and assessment, 1995.

[9] Wooldridge M. Jennings N.R., Sycara K. A roadmap of agent research and development. *Journal of Autonomous Agents and Multi-Agent Systems*, 1(1):7–38, 1998.

[10] Brodie S. Kennan P.B. A prototype web-based carpooling system. *Paper presented at the Americas Conference on Information Systems (AMCIS)*, Long Beach, California, USA, August 2000.

[11] Wooldridge M. *An introduction to MultiAgent Systems*. John Wiley and Sons, Chichester, 2002.

[12] Luigi F. Calvo R. Maniezzo V., Haastrup P. A distributed geographic information system for the daily car pooling problem. *Computers and Operations Research*, (In Press, Corrected Proof), July 2003.

[13] Spanoudakis N. I. Moraitis P., Petraki E. Engineering jade agents with the gaia methodology.

[14] Axtell R. Why agents? on the varied motivations for agent computing in the social sciences. *CSED Working Paper No. 17*, Published in Agent Simulation: Applications, Models and Tools, 1999, 11 2000.

[15] Sturm P. Rothkugel S. Carpacities: Distributed car pool agencies in mobile networks. *ASA/MA 2000*, pages 178–191, 2000.

[16] Sierra C. Sabater J. Social regret, a reputation model based on social relations. *SIGecom Exch.*, 3(1):44–56, 2002.

[17] URLFCNP. *FIPA - The Contract Net Protocol*. http://www.fipa.org/specs/fipa00029, (11/01/2004).

[18] URLHA. *UK Highways Agency real-time Traffic Monitoring System*. http://www.highways.gov.uk/news/m25_rt/index.htm, (12/01/2004).

[19] URLliftshare. *Liftshare.com - Major UK carpooling player*. http://www.liftshare.com, (12/01/2004).

[20] URLliftshareletter. *Liftshare - Sample Insurance Enquiry Letter*. http://www.liftshare.org/letter.htm, (12/01/2004).

[21] URLliftsharestats. *Liftshare System Statistics*. http://www.liftshare.co.uk/stats.asp, (12/01/2004).

[22] URLmapXML. *MultiMap XML Presentation - StoreFinder*. http://www.amitkoth.com/weblog/dissertation/documents/URLmapXML.pdf, Obtained from MultiMap 11/01/2004.

[23] URLMetronetLA. *Los Angeles County Ridesharing Programme*. url-http://www.mta.net/riding_metro/commute_services/carpques.html, (12/01/2004).

[24] URLNCS. *Nationalcarshare.co.uk*. http://www.nationalcarshare.co.uk/faq.html#links, (12/01/2004).

[25] URLNCSc. *Claims by National Car Share on the benefits of pooling in the UK*. http://www.nationalcarshare.co.uk/facts.html, (12/01/2004).

[26] URLSST. *The Seattle Smart Traveler Project*. http://www.its.washington.edu/projects/sst.html, (12/01/2004).

[27] URLSSTFTA. *Assessment of the Seattle Smart Traveler*. United States Department of Transportation: Federal Transit Administration (FTA), http://www.itsdocs.fhwa.dot.gov/jpodocs/repts_te/8r401!.pdf (12/01/2004).

[28] URLStatAuto. *StatAuto Car Sharing AG - in German*. http://www.statauto.de, Accessed 11/01/2004.

[29] URLTaxistop. *Taxistop Belgium, providing online and software route matching*. http://www.taxistop.be/4/4pool.htm, (12/01/2004).

[30] URLUBC. *Dynamic Ridesharing: Background and Options for UBC*. http://www.trek.ubc.ca/research/pdf/RidesharingReport_jul01.pdf, (12/01/2004) 2001.

[31] URLWCC. *World Carshare Consortium - An open platform for international collaboration*. http://www.worldcarshare.com/cs_index.htm, (12/01/2004).

[32] Maniezzo V. Exact and approximate nondeterministic tree-search procedures for the quadratic assignment problem, 1998.

[33] Krishnamoorthy M. Van Der Touw J. Estimating the viability of an instant car pooling system. *ASOR Bulletin*, 13(2), June 1994.

[34] Kinny D. Wooldridge M., Jennings N. The gaia methodology for agent-oriented analysis and design. *Autonomous Agents and Multi-Agent Systems*, 3(3):285–312, 2000.