

Using Role modelling and Synthesis to Reduce Complexity in Agent-Based System Design

A thesis submitted to the University of Manchester Institute
of Science and Technology for the degree of Doctor of
Philosophy

Anthony Karageorgos

Department of Computation

Manchester 2003

Declaration

No portion of the work referred to in the thesis has been submitted in support of an application for another degree or qualification of this or any other university, or other institution of learning.

" Όσα βουνά κι άν ανεβείτε
απ' τις κορφές τους θ' αγναντεύετε
άλλες κορφές, ψηλότερες
μιαν άλλη πλάση, ξελογιάστρα.

Και στην κορφή σα φτάστε την κατάψηλη,
πάλι θα καταλάβετε πως βρίσκεστε
σαν πρώτα, κάτω απ' όλα τ' άστρα..."

Κωστής Παλαμάς (1859-1944)
Έλληνας ποιητής

"As many mountains as you climb
from their peaks you will be gazing
some other peaks, higher
another world, enchanting.

And when you reach the very highest peak,
again you realize that you lie
under all stars, as before..."

Costis Palamas (1859-1944)
Greek Poet

Acknowledgements

First of all, I would like to thank my supervisor, Nikolay Mehandjiev, for his guidance and cooperation on my research efforts in this PhD project. Nikolay has offered particularly insightful comments and important feedback during our regular discussions held at the round table in his office. Furthermore, Nikolay has always been available for providing mentoring and support in all matters that could possibly concern a person carrying out a PhD.

This work would not have been as successful if I had not had the opportunity to cooperate with Simon Thompson from BT Exact Technologies. Simon was not only an excellent research collaborator, but he has also shown a unique attitude for cooperation providing support, encouragement and ideas. As a result, doing research with Simon was a pleasure rather than work, regardless of the research challenges that had to be faced.

UMIST provided a comfortable and stimulating environment for research. I found UMIST ideal for combining research findings and technological innovations and applying them to real world problems in the industry.

This work has been funded by BT Laboratories (currently BT Exact Technologies). I have spent some quite productive time there aligning my research with that of large, relevant BT projects and collaborating with researchers with similar interests. The outcome of my visits to BT was really fruitful thanks to all those that I have interacted with.

Furthermore, I would like to thank my wife, Eleni, who has been patient and encouraging all these years when I was stealing time from her to put it into research. Eleni's support made all difficulties in life look simpler and certain to be resolved.

Last, but not least, I would like to thank my parents who have always advised me to follow my dreams and who have always supported my decisions and choices, even if they had different personal preferences.

Abstract

Real-world applications of Agent-Based Systems (ABS) tend to involve significant design complexity, making the activities of understanding and manipulating the concepts and models needed for the detailed design increasingly difficult. Reducing this complexity is the main aim of the PhD work reported here.

Complexity in software design can be reduced by increasing the level of abstraction where design decisions are taken and by executing a number of design process steps automatically. Along this line, the main research objectives of this project are to (a) introduce appropriate concepts and techniques to allow designers to work at a high level of abstraction and (b) to develop a design process that allows semi-automated progress from analysis to design. The first objective involves the need to consider agent organisational settings and collective behaviour as first class design constructs, whilst the second requires support for taking non-functional aspects and software design heuristics into account in design decisions.

The main contribution of this thesis is the proposed Role-Algebraic Multi-Agent System Design (RAMASD) method, which reduces ABS design complexity by enabling designers to work at a high level of abstraction and by semi-automating the design process. To meet the above objectives, RAMASD models agent behaviour using roles and it views ABS design as a problem of allocating roles to appropriate agents. RAMASD represents all design requirements by using appropriate roles and constraints on role characteristics. Two innovative ideas behind RAMASD are to enable high-level design by defining the role concept so that it can represent a rich set of agent behavioural aspects and to use the synthesis concept as the basis for semi-automating the design process.

These two ideas are supported by the main innovation of RAMASD, the *role algebra*. The role algebra is a formal model of role relations concerning allocation of roles to agents. The semantics of this model are described using a two-sorted algebra. The role algebra leverages both high-level design, enabling specification of design constraints at the role level, and semi-automation of the design process by enabling automatic role allocation after role selection has been made.

An extension to the Zeus agent building toolkit has been constructed to implement support for RAMASD. To test the applicability of RAMASD, it has been applied in two case studies. The value of RAMASD in regards to reducing complexity has been shown by comparing it with similar methods using a design complexity evaluation framework. This is followed by a detailed comparison with Gaia, a representative ABS design method, in the context of a case study. In all cases, the superiority of RAMASD is clearly demonstrated.

Publications, Presentations and Patents

(based on the work described here)

1. Karageorgos, A., S. Thompson and N. Mehandjiev, 2003. "Specifying Reuse Concerns in Agent System Design Using A Role Algebra". In Kowalczyk, R., Müller, J., Tianfield, H. and Unland, R., eds. *Agent Technologies, Infrastructures, Tools, and Applications for e-Services*. LNAI 2592, Berlin-Heidelberg: Springer Verlag, p. 121-136, ISBN 3-540-00742-3.
2. Karageorgos, A., N. Mehandjiev and S. Thompson, 2002. "RAMASD: A Semi Automatic Method for Designing Agent Organisations", *Knowledge Engineering Review, Special Issue on Coordination and Knowledge Engineering*, 17, 4 (Fall 2002), 57-84.
3. Karageorgos, A., S. Thompson and N. Mehandjiev, 2002. "Agent-based system design for B2B electronic commerce", *International Journal of Electronic Commerce, Special Issue on Agent Technologies for B2B Electronic Commerce*, 7, 1, (Fall 2002), 59-90.
4. Karageorgos, A., S. Thompson and N. Mehandjiev, 2002. "Specifying Reuse Concerns in Agent System Design Using A Role Algebra". *Workshop on Agent Technologies for e-services (ATES 2002), held in conjunction with Net.ObjectDays (NODE 2002)*. Erfurt, Germany, October 7-10.
5. Karageorgos, A., S. Thompson and N. Mehandjiev, 2002. "Semi-Automatic Design of Agent Organisations". In *Proceedings of ACM Symposium in Applied Computing, Special Track on Coordination Models, Languages and Applications*, Madrid, Spain, March 10-14, ACM Press, pp. 306-313.
6. Karageorgos, A., S. Thompson and N. Mehandjiev, 2001. "Designing Agent Systems using a Role Algebra", Fourth Workshop of the UK Special Interest Group on Multi-Agent Systems, Oxford, UK, December 13-15, UKMAS SIG.
7. Thompson, S. and A. Karageorgos 2001. "Multi-Agent System Design Using Role Models". *Patent Application*. BT Case Ref A26117, UK, (Autumn 2001).

Table of Contents

CHAPTER 1.....	1
INTRODUCTION	1
1.1 MOTIVATION	1
1.2 CONTEXT OF THE THESIS	2
1.3 ISSUES AND CHALLENGES	3
1.3.1 <i>The ABS Design Complexity Problem</i>	3
1.3.2 <i>Reusing Design Knowledge</i>	4
1.3.3 <i>Non-Functional Aspects and Design Heuristics</i>	4
1.4 AIMS AND OBJECTIVES	5
1.5 MAIN CONTRIBUTIONS	6
1.6 RESEARCH METHODOLOGY	7
1.7 CASE STUDY DESCRIPTIONS	8
1.8 THESIS ORGANISATION.....	9
CHAPTER 2.....	11
AGENT-BASED SYSTEM DESIGN	11
2.1 DESIGNING ABSS	11
2.2 OVERVIEW OF AGENT CONCEPTS.....	11
2.2.1 <i>Agent-Oriented vs Object-Oriented Approaches</i>	11
2.2.2 <i>Defining the Term ‘Agent’</i>	13
2.2.3 <i>A Simple Agent Formal Model</i>	15
2.2.4 <i>An Example of a Simple Agent</i>	15
2.2.5 <i>Using Roles to Model Agent Behaviour</i>	16
2.2.6 <i>Agent Architecture</i>	17
2.3 AGENT-BASED SYSTEMS	19
2.3.1 <i>Overview</i>	19
2.3.2 <i>Interaction in ABSs</i>	20
2.3.3 <i>ABSs as Organisations of Agents</i>	20
2.3.4 <i>Software Complexity and ABS Design</i>	21
2.3.4.1 Complexity in Software Engineering.....	21
2.3.4.2 Complexity in ABS Design.....	22
2.4 CLASSIFICATION OF ABS ENGINEERING APPROACHES.....	24

2.4.1	<i>Ad-hoc Approaches</i>	25
2.4.2	<i>Formal Approaches</i>	25
2.4.3	<i>Informal Approaches</i>	26
2.4.3.1	Approaches Based on Object-Oriented Software Engineering	27
2.4.3.2	Approaches Based on Information Systems Engineering.....	27
2.4.3.3	Approaches Based on Knowledge Engineering.....	28
2.4.3.4	Tool-Based Approaches.....	28
2.4.4	<i>Dynamic Approaches</i>	29
2.4.5	<i>Overall Assessment</i>	29
2.5	SUMMARY	31

CHAPTER 3..... 33

ASSESSMENT OF ABS ENGINEERING APPROACHES33

3.1	AN EVALUATION FRAMEWORK FOR ABS DESIGN.....	33
3.1.1	<i>Concepts</i>	34
3.1.2	<i>Models</i>	35
3.1.3	<i>Process</i>	36
3.1.4	<i>Pragmatics</i>	37
3.2	COMPARATIVE EVALUATION OF ABS ENGINEERING APPROACHES	40
3.3	IMPLICATIONS FOR FURTHER RESEARCH	41
3.3.1	<i>Support for Design Heuristics</i>	42
3.3.2	<i>Organisational Settings</i>	42
3.3.3	<i>Collective Behaviour</i>	43
3.3.4	<i>Non-Functional Aspects</i>	44
3.3.5	<i>Automating the Design Process</i>	45
3.3.6	<i>Working at Different Abstraction Levels</i>	45
3.4	SUMMARY	47

CHAPTER 4..... 49

USING ROLE MODELLING FOR ABS DESIGN49

4.1	COMPLETE ROLE MODELLING APPROACHES.....	49
4.2	MODELLING SOCIAL BEHAVIOUR USING ROLES	50
4.2.1	<i>Defining the Term ‘Role’: a Social View</i>	50
4.2.1.1	Social Aspects of Role Definitions.....	50
4.2.1.2	Role Relationship Zones.....	51
4.2.2	<i>Overview of Role Theory</i>	52

4.2.3	<i>Role Theoretic Concepts</i>	53
4.2.4	<i>Role Dependency Relations</i>	55
4.2.5	<i>Role Identification and Role Types</i>	56
4.3	USING ROLES IN INFORMATION SYSTEMS MODELLING	57
4.3.1	<i>Roles in Business Systems Modelling</i>	57
4.3.2	<i>Role-Based Access Control in Distributed Systems Management</i>	59
4.3.3	<i>Roles in Object Oriented Software engineering</i>	60
4.3.3.1	Defining Roles in Object Oriented Software Engineering	61
4.3.3.2	Role Properties	63
4.3.3.3	Role Relationships, Synthesis and Synergy	65
4.3.4	<i>Roles in ABS Modelling</i>	66
4.3.4.1	Modelling Goal-Based Interactions Using Roles	67
4.3.4.2	Modelling Organisational Settings Using Roles	68
4.4	USING ROLES FOR THE DESIGN OF ABSS	69
4.4.1	<i>Comparison of Role Modelling Approaches</i>	69
4.4.2	<i>Formalising Role Dependency Relations</i>	71
4.5	SUMMARY	71

CHAPTER 5..... 73

THE RAMASD METHOD.....73

5.1	USING ROLE MODELLING AND SYNTHESIS FOR ABS DESIGN	73
5.2	ROLE MODELLING IN RAMASD	74
5.2.1	<i>Defining Roles and Role Models</i>	74
5.2.1.1	Role Characteristics	74
5.2.1.2	Properties of Roles and Role Models	76
5.2.2	<i>Representing and Using Role Models</i>	77
5.2.3	<i>Role Model Types</i>	77
5.2.4	<i>Identification of Roles in the Application Domain</i>	78
5.2.4.1	Criteria for Role Identification	78
5.2.4.2	Goal-Oriented Role Identification	79
5.2.4.3	Role Identification for an e-Business Security System	82
5.2.5	<i>Management of the Role Modelling Process</i>	84
5.2.6	<i>Consistency of Role-Based Specifications</i>	85
5.2.7	<i>Rigorous Role Assignment Using Role Algebra</i>	85
5.2.7.1	Relations in the Role Algebra	86
5.2.7.2	Semantics of the Role Algebra	88
5.2.7.3	Graphical Representation of Role Relations	90
5.3	APPLYING THE SYNTHESIS CONCEPT TO ABS DESIGN	90

5.3.1	<i>Synthesis in Traditional Engineering</i>	90
5.3.2	<i>A Synthesis-Based Design Process Model</i>	92
5.4	THE RAMASD DESIGN PROCESS	95
5.5	THE INNOVATIVE FEATURES OF RAMASD.....	97
5.5.1	<i>The Philosophy of the RAMASD Approach</i>	97
5.5.2	<i>Reusing Collective Behaviour</i>	97
5.5.2.1	Representing and Using Patterns of Behaviour	97
5.5.2.2	An Example of Behaviour Reuse	98
5.5.3	<i>Representing Organisational Settings</i>	100
5.5.4	<i>Considering Non-Functional Aspects</i>	102
5.5.5	<i>Considering Design Heuristics</i>	105
5.6	USING RAMASD WITH EXISTING METHODOLOGIES.....	107
5.7	SUMMARY	108

CHAPTER 6.....109

IMPLEMENTATION OF RAMASD.....109

6.1	EXTENDING ZEUS TO SUPPORT RAMASD	109
6.2	THE ZEUS AGENT BUILDING TOOLKIT	111
6.3	EXTENDING ZEUS TO SUPPORT RAMASD	111
6.3.1	<i>The Extended Zeus Agent Development Methodology</i>	111
6.3.2	<i>The Extended Zeus Visual Agent Creator Component</i>	113
6.4	THE RCL CONSTRAINT LANGUAGE	117
6.5	ALLOCATING ROLES TO AGENTS	118
6.6	SUMMARY – CONCLUSIONS	120

CHAPTER 7.....121

CASE STUDIES: MOBILE WORKFORCE SUPPORT AND COVISINT.....121

7.1	APPLYING RAMASD TO REAL WORLD CASES	121
7.2	MOBILE WORKFORCE SUPPORT	122
7.2.1	<i>The Mobile Workforce Support Problem</i>	122
7.2.2	<i>Role Identification</i>	123
7.2.3	<i>Specifying Design Constraints</i>	126
7.2.4	<i>Design Results</i>	127
7.3	EXAMPLE: AN AUTOMOTIVE INDUSTRY B2B EXCHANGE	129
7.3.1	<i>Case Study Overview</i>	129
7.3.2	<i>Role Identification</i>	132

7.3.3	<i>Qualitative Modelling of Non-Functional Aspects</i>	135
7.3.3.1	Security Issues	135
7.3.3.2	Privacy Issues	136
7.3.4	<i>Organisational Settings</i>	137
7.3.5	<i>Role Composition</i>	138
7.3.6	<i>Specifying Design Constraints</i>	139
7.3.7	<i>Role Allocation Results</i>	141
7.4	SUMMARY – CONCLUSIONS	143
CHAPTER 8.....		145
EVALUATION OF RAMASD.....		145
8.1	SELECTING AN EVALUATION APPROACH	145
8.1.1	<i>Approaches to Evaluating Software Engineering Methods</i>	145
8.1.2	<i>Evaluating RAMASD</i>	146
8.1.3	<i>Selecting Case Studies and Test Scenarios</i>	147
8.2	FRAMEWORK-BASED EVALUATION	147
8.2.1	<i>Main Features of RAMASD</i>	147
8.2.2	<i>Comparing RAMASD With Other Methods</i>	149
8.3	COMPARISON OF RAMASD AND GAIA	151
8.3.1	<i>Overview of Gaia</i>	151
8.3.2	<i>Applying Gaia in the Mobile Workforce Case Study</i>	152
8.3.3	<i>Limitations of Gaia</i>	155
8.4	DISCUSSION	158
8.4.1	<i>Real World Applicability of RAMASD</i>	158
8.4.1.1	The Generality of RAMASD	158
8.4.1.2	The Scalability of RAMASD	159
8.4.2	<i>Novel Aspects of RAMASD</i>	159
8.4.2.1	The Innovative Features of RAMASD	159
8.4.2.2	The Role Algebra	160
8.5	SUMMARY	161
CHAPTER 9.....		163
CONCLUSIONS.....		163
9.1	REVISITING THE RESEARCH HYPOTHESIS	163
9.2	ASSESSING THE THESIS CONTRIBUTIONS	164
9.3	LIMITATIONS OF RAMASD	165

9.4	FURTHER WORK.....	166
9.5	CONCLUDING REMARKS	167
APPENDICES.....		169
APPENDIX A EVALUATION OF ABS DESIGN APPROACHES		169
A.1	RAPPID	169
A.1.1	Overview of RAPPID	170
A.1.2	Evaluation of RAPPID	171
A.1.3	Strengths and Weaknesses of RAPPID.....	172
A.2	DESIRE	172
A.2.1	Overview of DESIRE.....	174
A.2.2	Evaluation of DESIRE.....	174
A.2.3	Strengths and Weaknesses of DESIRE.....	175
A.3	GAIA	175
A.3.1	Overview of Gaia.....	175
A.3.2	Evaluation of Gaia.....	176
A.3.3	Strengths and Weaknesses of Gaia	177
A.4	TROPOS	177
A.4.1	Overview of Tropos.....	177
A.4.2	Evaluation of Tropos.....	178
A.4.3	Strengths and Weaknesses of Tropos	179
A.5	MESSAGE	179
A.5.1	Overview of MESSAGE/UML.....	179
A.5.2	Evaluation of MESSAGE.....	182
A.5.3	Strengths and Weaknesses of MESSAGE/UML	183
A.6	ZEUS.....	183
A.6.1	Overview of Zeus Agent Development Methodology	183
A.6.2	Evaluation of Zeus Agent Development Methodology	185
A.6.3	Strengths and Weaknesses of Zeus.....	186
A.7	KARMA/TEAMCORE.....	187
A.7.1	Overview of KARMA/TEAMCORE.....	187
A.7.2	Evaluation of KARMA/TEAMCORE.....	189
A.7.3	Strengths and Weaknesses of KARMA/TEAMCORE	189
APPENDIX B THE ZEUS TOOLKIT.....		191
B.1	THE COMPONENTS OF THE ZEUS TOOLKIT	191
B.1.1	The Agent Component Library	191

<i>B.1.2</i>	<i>The Visualisation Tools</i>	192
<i>B.1.3</i>	<i>The Agent Building Tools</i>	192
B.2	THE ZEUS AGENT SYSTEM REALISATION PROCESS	195
B.3	THE ZEUS UTILITY AGENTS	198
B.4	THE GENERIC ZEUS AGENT.....	200
APPENDIX C RCL EBNF SYNTAX.....		203
REFERENCES		207

Table of Figures

Figure 1.1: PhD research question and solution approach	6
Figure 1.2: Thesis organisation.....	9
Figure 2.1: Perceive-Reason-Act cycle.....	13
Figure 2.2: A simple agent formal model.....	14
Figure 2.3: A container terminal yard agent.....	15
Figure 2.4: Agent internal components	18
Figure 2.5: An agent organisation.....	20
Figure 2.6: Classification of ABS engineering approaches.....	24
Figure 3.1: A framework for comparing ABS engineering approaches with respect to design ..	34
Figure 4.1: Role relationship zones	52
Figure 4.2: Agent-Position-Role dependencies in the Actor-Dependency model	58
Figure 4.3: Role characteristics for distributed systems access control	60
Figure 4.4: Roles as association names	61
Figure 4.5: Roles as patterns of behaviour	61
Figure 4.6: Object–Role relationships (Wong 1997)	63
Figure 4.7: The Bureaucracy pattern represented as a role model (Richle 1997).....	64
Figure 4.8: Sample RRC card for the Bureaucracy pattern (Kendal 1999).....	65
Figure 4.9: A high level view of the supply chain management role model (Kendal 1999).....	66
Figure 4.10: An example MASE role model (DeLoach et., al. 2001).....	67
Figure 5.1: Schematic representation of a role model using UML notation.....	77
Figure 5.2: The phases of a goal-oriented role identification method.....	80
Figure 5.3: Goal cases for an e-business security protection system	82
Figure 5.4: Goal hierarchy tree and role identification for an e-business security system.....	83
Figure 5.5: Identified roles for the e-business security system.....	84
Figure 5.6: Semantics of the role algebra.....	88
Figure 5.7: Graphical notation for the relations of the role algebra.....	90
Figure 5.8: The synthesis problem solving process	91
Figure 5.9: A generic synthesis-based design process model.....	93
Figure 5.10: Schematic representation of the RAMASD design process.....	95
Figure 5.11: An example of collective behaviour reuse in RAMASD	98
Figure 5.12: RAMASD roles for the conference management system example	99
Figure 5.13: Enforcing organisational rules by appropriate merging of roles	101
Figure 5.14: An example of modelling organisational rules using roles.....	102
Figure 5.15: Extended actor diagram for an e-cultural system (aft Giorgini et al., 2001)	103

Figure 5.16: Using the FIPA directory facilitator role model for e-culture service brokering..	104
Figure 5.17: A personal assistant role model	105
Figure 5.18: Spheres of responsibility (Collins et al. 1999).....	106
Figure 5.19: Specifying the spheres of responsibility heuristic using role relations.....	107
Figure 6.1: Conceptual view of the extended Zeus ABS design tool	110
Figure 6.2: The extended Zeus agent development methodology	112
Figure 6.3: The extended Zeus Agent Generator component.....	113
Figure 6.4: The role model and role definition editors	114
Figure 6.5: The Role Constraints Editor component	115
Figure 6.6: The Role Allocation component	116
Figure 6.7: Parts of an RCL specification	118
Figure 6.8: A simple search algorithm for allocating roles to agent types	119
Figure 7.1: A high level view of the mobile workforce coordination case study	122
Figure 7.2: Use case goals for the telephone repair service teams case study.....	124
Figure 7.3: Role models for the telephone repair service teams case study	125
Figure 7.4: Compositional constraints for the telephone repair service teams case study	126
Figure 7.5: Snapshot of the extended Zeus toolkit for the mobile workforce case study.....	128
Figure 7.6: Agent types for the telephone repair service teams case study	129
Figure 7.7: Use case goals for an automotive industry B2B exchange case study	131
Figure 7.8: Role models for the automotive industry B2B exchange case study	133
Figure 7.9: The mediator pattern	137
Figure 7.10: Updated role models based on the mediator pattern	138
Figure 7.11: Compositional constraints for the B2B exchange case study	140
Figure 7.12: Agent types for the B2B exchange case study	142
Figure 7.13: Snapshot of the extended Zeus toolkit for the B2B exchange case study.....	143
Figure A.1: The RAPPID ABS architecture.....	170
Figure A.2: A generic agent model in DESIRE.....	172
Figure A.3: Relations between Gaia models	175
Figure A.4: Knowledge level concepts in MESSAGE/UML	180
Figure A.5: The Zeus agent development methodology	184
Figure A.6: The Zeus agent architecture.....	185
Figure A.7: The KARMA/TEAMCORE Framework.....	187
Figure B.1: The components of the Zeus agent building toolkit (Collins et al. 1999).....	192
Figure B.2: The Control and Society Tools of the Zeus agent building toolkit.....	193
Figure B.3: The agent definition interface of the Zeus agent building tool	193
Figure B.4: The Ontology editor.....	194
Figure B.5: The Zeus agent realisation process (Nwana et al. 1999).....	195

Figure B.6: The flow of information between an agent and a task	197
Figure B.7: The Zeus ABS structure (Thompson 2001)	199
Figure B.8: The generic Zeus agent internal structure (Collins and Ndumu 1999)	201

List of Tables

Table 3.1: Description and ranking of evaluation framework aspects	39
Table 3.2: Comparison of ABS engineering approaches	40
Table 4.1: Strengths and weaknesses of role modelling approaches	70
Table 5.1: Role characteristics.....	74
Table 8.1: Comparing RAMASD with other ABS design methods	150
Table 8.2: The role schema for the REPAIR_WORKER role	153
Table 8.3: Role schemata for the MANAGER and CUSTOMER_HANDLER roles	154
Table 8.4: Role schemata for the TRAVEL_DEPT and EXPERTISE_KNOWLEDGE roles ..	155
Table A.1: Evaluation of RAPPID.....	171
Table A.2: Evaluation of DESIRE.....	174
Table A.3: Evaluation of Gaia.....	176
Table A.4: Evaluation of Tropos	178
Table A.5: Evaluation of MESSAGE/UML.....	182
Table A.6: Evaluation of the Zeus agent development methodology	186
Table A.7: Evaluation of KARMA/TEAMCORE.....	189

Chapter 1

Introduction

This chapter describes the context of this work, starting with the main challenges in Agent-Based System (ABS) design. Then the hypothesis, aims and objectives of the research work undertaken are presented, followed by a discussion regarding the novelty of this research and its contributions. Finally, a description of the case studies used in the evaluation and the structure of the thesis is given.

1.1 Motivation

Agent-based applications are developed to address the need for software operating in open and dynamically changing environments, such as the Internet, and they use the key abstraction of a software *agent*. Agents are software components that are situated in an environment, are able to act autonomously, reactively and proactively and have social abilities [211]. An agent-based application normally includes more than one agent referred to as an *Agent-Based System (ABS)* or *Multi-Agent System (MAS)* [96].

ABSs typically contain many dynamically interacting agents, each with their own thread of control, that engage in complex interactions with each other and their environment. Such systems can be considerably more difficult to correctly and efficiently design than those that simply compute a function of some input through a single thread of control.

Existing approaches to ABS design leave most of the design decisions to the designer, who has to tackle the system complexity based on intuition and experience. The main reason for this is that current ABS design methods do not provide the necessary models, process steps and formal mechanisms that would allow reducing the complexity involved by enabling work at a high-level of abstraction and by semi-automating the design process. Developing a new ABS design method that addresses ABS design complexity by handling these issues was therefore selected to be the topic of this PhD.

The research reported here has been conducted under the assumption that agent-based software will increasingly be used to address the dynamism and the openness of contemporary software environments. To enable this widespread use, the complexity of ABS design should be effectively addressed.

1.2 Context of the Thesis

ABSs can currently be designed using a number of ad-hoc methods, formal methods or informal but structured methods. In addition, design can be done either statically, before the ABS is deployed, or dynamically at run-time. All existing methods have certain weaknesses and involve considerable difficulty in understanding and manipulating the concepts and models needed for the detailed ABS design. This is referred to as *design complexity*. This PhD proposes an informal and structured method which addresses the design complexity problem by semi-automating the design process and by enabling design at high levels of abstraction. This research does not consider ABS design on run-time.

Due to their special properties, software agents cannot be effectively designed by directly applying traditional software design methods. In particular, agents are not simple procedures and hence they cannot be designed by traditional methods for designing procedural software. Furthermore, agents supersede objects since, unlike objects, they can control their state. For example, agents autonomously decide if a particular functionality is in accordance to their current state and goals before executing it, whilst objects simply execute their public methods when these are invoked. Object-oriented design methods therefore are also not suitable for the design of ABSs.

The unsuitability of traditional software design methods has spawned new methods specifically targeting ABS design. These can be ad-hoc, formal, informal and structured, and dynamic. *Ad-hoc* design involves designing an ABS in an application domain specific manner [34]. Ad-hoc designs are difficult to justify, evaluate and systematically improve. *Formal* ABS design approaches are based on the use of formal methods [210]. Formal methods enable specification of agent behaviour in a rigorous manner. However, they suffer from significant drawbacks. For example, there is usually no precise relationship between the abstractions used in the specification model and any concrete computational model. Therefore, *informal* and *structured* methods have emerged. These methods originate from knowledge engineering and software engineering and are predominantly based on object-oriented analysis and design methodologies. Finally, dynamic methods involve defining the structure of an ABS and the behaviour of the individual agents *dynamically* on run-time [9, 76, 93, 192] but they are resource consuming and may result in unstable systems [185]. Informal and structured methods are regarded as practical for numerous real-world applications [208] and therefore they form the context of this work.

Based on the view that design complexity decreases with increasing the level of abstraction and with semi-automating the design process [1], the efforts in this work concentrated on addressing these two issues. To this end, a powerful modelling concept was required to represent agent behaviour and the *role* concept has been selected as such. Roles refer to encapsulated behaviour

corresponding to positions in organisations or parties in interactions. Therefore, roles are particularly suitable for modelling agent behaviour and they are used in the majority of ABS engineering methodologies, for example in [30, 64, 108, 209]. Role modelling enables the representation of agent behaviour by a number of roles assigned to an agent. In that case, designing a multi-agent system refers to identifying and assigning appropriate roles to agents. This research follows this approach by considering roles as the primary constructs for ABS design. In particular, emphasis is given to formally describing the interrelations between roles as far as it concerns their assignment to agents. This formalisation enables executing some of the design steps automatically and it also increases the level of abstraction where design constraints are specified.

1.3 Issues and Challenges

There are currently many challenges in ABS design. The focus of this work is on a problem often identified as a core challenge [1, 186]: reducing the complexity the designers must handle. The solution given is aligned with current research trends with respect to a number of important design issues: Organisational settings and collective behaviour are considered first class design constructs and design heuristics and non-functional aspects are taken into account in design decisions.

1.3.1 The ABS Design Complexity Problem

In order for ABSs to be effective in real world applications, they must be reliable and robust. Designing ABSs is a non-trivial task. Given a set of analysis models, design involves decisions regarding the amount of intelligence of each agent, the properties that each agent will have, the lines of inter-agent communication and the authority relationships between agents. Since agents are considered autonomous entities, many researchers use the terms *agent organisation* to refer to an ABS [64, 185, 209] and *organising* to refer to the process of creating an agent organisation. There is no generic solution to an organising problem in the sense that there is no best organisation for all situations [46, 185]. Therefore, the aim is to find a satisfactory solution, for example a solution where the designed ABS satisfies all application requirements and design constraints.

Currently, ABS design methods delegate the decision-making responsibility regarding all aspects of design to the designer, who handles the resultant system complexity based on creativity and intuition. Design decisions are therefore exposed to human error. Semi-automating transformation from analysis to design and increasing the level of abstraction where design decisions are taken are considered necessary to address this problem [186]. Indeed, automatic transformations of at least some aspects of the analysis models to design and

providing high-level design constructs can change this, allowing designers focus on aspects where their creativity is truly necessary. In the case of role-based design, for example, the designer may select the appropriate role models and an automation tool will carry out the allocation of roles to agents.

1.3.2 Reusing Design Knowledge

There is a consensus that reusing design knowledge reduces design complexity allowing designers to work with concepts of larger granularity at higher abstraction levels. In ABS design this refers to reusing knowledge about goal-driven behaviour of agents. A major challenge in this respect is how to integrate the different types of reused behaviours in a seamless manner.

Reusing design knowledge has been identified as a technique to manage software complexity and reduce cost and time to market of software products [1]. Examples of attempts to address this issue include reusing conceptual models (design patterns) [41] and reusing design specifications [124]. In particular, much work is expended in discovering patterns in various domains. However, techniques to deploy these proven design knowledge reuse solutions in ABS design are still lacking effective support since efficient composition mechanisms to glue patterns together at the design level do not exist yet [214].

In ABS engineering literature, design knowledge can refer to agent application functionality [108, 150] and to agent organisational settings [64]. The term *organisational settings* covers the general rules and conventions between entities in an organisation as well as various authority relationships and coordinating interactions among these entities [73]. Reused agent application behaviours correspond to *collective behaviour patterns* [108], while widely used agent organisational settings that are applicable to many types of ABSs are termed *organisational patterns* [220]. Both types of patterns play an important role in ABS design. If patterns are to be helpful when implementing, a way to integrate them is required. In role modelling, design patterns can be represented by appropriate role models. Hence, efficient techniques for allocating roles to agents considering role interrelations are needed.

1.3.3 Non-Functional Aspects and Design Heuristics

In addition to application functionality, ABS design must consider non-functional aspects, for example security and performance. Furthermore, design heuristics should be able to be applied in a systematic manner to construct satisfactory designs. Non-functional aspects and design heuristics specify additional design constraints increasing the design complexity. Hence, to reduce design complexity, effective techniques for automatically taking non-functional aspects and design heuristics into account in design are required,

The complexity of the agent properties adds difficulty to the problem of designing agent behaviour, whilst achieving particular non-functional qualities. Existing approaches to ABS design do not address this issue although the case for considering non-functional aspects in ABS design has been raised [60]. When ABS design is based on role modelling, non-functional aspects need to be represented within role models and be considered when allocating roles to agents. To reduce design complexity this has to be done in an automatic manner.

The quality of software designs can be improved by applying software design heuristics. Such heuristics can be either general, for example low cohesion and high coupling [118], or specific to ABS design. For example, it has been suggested that the behaviour responsible for handling a system resource should be allocated to one agent only [38]. Design heuristics should be supported by an effective method for semi-automatic ABS design. Appropriate mechanisms for automatically handling them during ABS design are thus required.

1.4 Aims and Objectives

Given the above context for ABS design and the challenges identified, the overall aim of the work described in this thesis is to develop a method for designing ABSs which reduces the level of design complexity compared to existing methods. The method developed is based on the premise that ABS design concerns the allocation of a set of roles R to a set of agents A such that the resulting design satisfies the application requirements.

The main approach to design complexity reduction pursued in this thesis includes increasing the level of abstraction during design and semi-automating the design process. These involve considering collective behaviour and organisational settings as first class design constructs and taking non-functional aspects and design heuristics into account in an automatic manner. The approach is enabled by formalising relations among roles to facilitate assignment of roles to agents, and by applying the synthesis concept to the design process. The approach is shown in Figure 1.1. In particular, the following hypothesis is made:

Hypothesis: *Formalising role relations in a formal algebraic model (the role algebra) and developing a synthesis-based design process can assist in developing an ABS design method, which reduces complexity in ABS design.*

To this end, the objectives of this research are:

1. To identify a basic set of possible relations among roles which specify inter-role constraints related to the process of assigning roles to agents, and introduce a formal model describing those relations. This formal model will both increase abstraction level and enable automatic allocation of roles to agents.

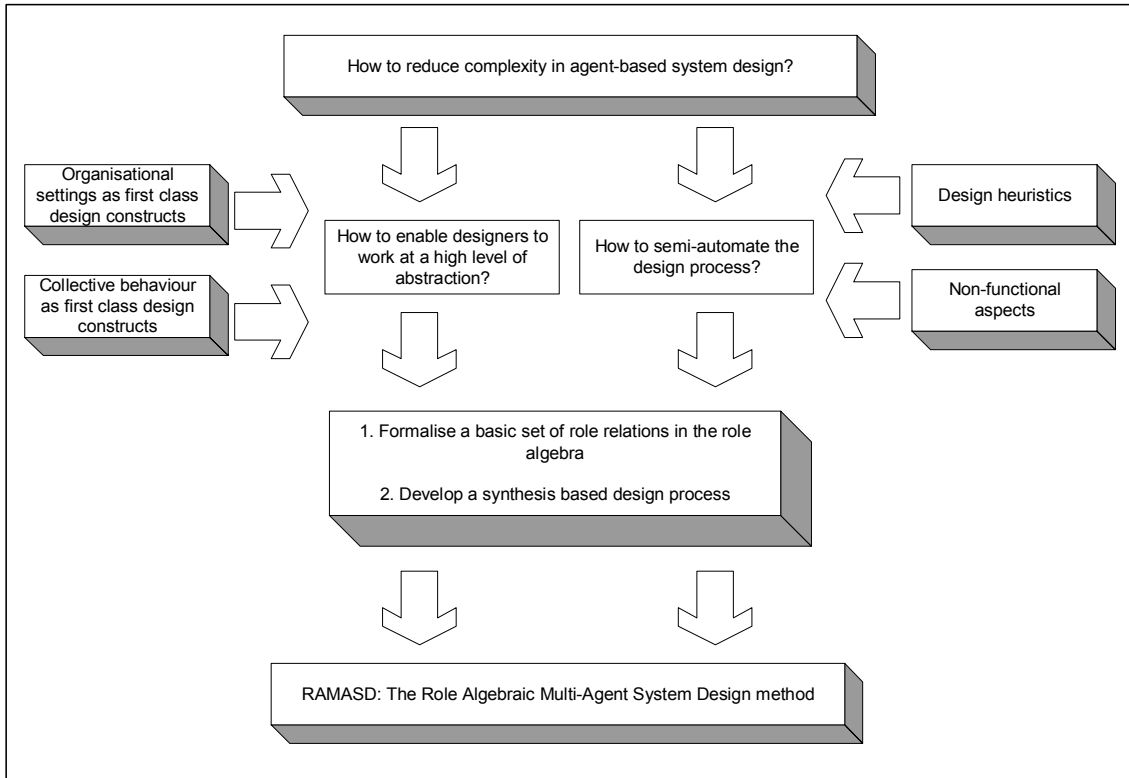


Figure 1.1: PhD research question and solution approach

2. To develop a semi-automatic role-based method for ABS design. This involves describing the steps, both manual and automatic, that need to be taken to produce an ABS design model. Furthermore, this requires introducing appropriate techniques that can be used to incorporate non-functional aspects and design heuristics in role models. Non-functional aspects and design heuristics can be treated as constraints affecting allocation of roles to agents in addition to the constraints described by role relationships.
3. To integrate the proposed method in an ABS design tool, which can be used to assist the user in applying the method.
4. To prove that the approach is feasible by applying it to a number of case studies. This will be done using the ABS design tool, which will have been previously implemented.
5. To evaluate the utility of the proposed method by comparing it with existing ABS design methods as far as it concerns design complexity.

1.5 Main Contributions

The overall contribution of this work is the RAMASD ABS design method and its value is assessed in Chapter 8. The development of RAMASD, however, involved a number of research tasks, which resulted in the following additional contributions:

1. A classification and comprehensive evaluation of current ABS design methods with regard to design complexity.
2. A formal algebraic model (the role algebra) describing relations among roles as far as it concerns assignment of roles to agents.

The contributions of this work have been under successful peer review and presented in refereed conferences [102] and published in accredited journals [103, 104]. Furthermore, the role algebra underpinning the RAMASD method is the subject of a patent application of the author and BT labs [101], which has so far proceeded to the final stages. Finally, BT are currently planning a commercial exploitation of RAMASD in a forthcoming commercial version of the Zeus agent building toolkit [147].

1.6 Research Methodology

This PhD work spans different research areas including Formal Methods, Organisational and Social Theories and Software Engineering. To identify relations among roles, it uses principles of human organisation. A formal model of role relations and a synthesis-based design process are then developed. The next steps of the research methodology concern developing an ABS design method based on that model, implementing the method in a tool and conducting experiments to test the effectiveness of the method in different ABS design scenarios.

Existing role-based approaches to ABS design stress the need to identify and characterise relations between roles [2, 107, 168]. However, only a small number of approaches investigate the consequences of role relations on ABS design, e.g. [107]. This is partly due to lack of formal foundations of role relations. Therefore, in this work role relations that would affect ABS design have been identified and formalised in an algebraic specification model. Role identification used *role theory* [15] and other organisational principles.

In this thesis, role relations existing in human organisations have been analysed with the aim of using them to specify agent behaviour. This exploits the traditional bias of ABS research towards modelling the way in which human organisations work. Indeed, roles have been extensively used in human organisations [126, 219] and application of roles to ABS design was seen as a natural progression.

A formal and rigorous description of role relations is necessary for semi-automatic ABS design. This was provided by a formal algebraic model of role relationships concerning assignment of roles to agents. Furthermore, formal underpinnings have to be combined with a systematic problem solving approach to semi-automate the design process. Such an approach would address the NP-hard problem of finding a satisfactory design solution among all design alternatives.

A design process based on the synthesis problem solving approach is appropriate. It involves decomposing the initial problem into sub-problems, independently solving them and integrating the sub-solutions into an overall solution while various constraints within and among sub-solutions are observed [4]. This approach leads to a semi-automatic design process which is able to find a satisfactory ABS design solution, if it exists.

The RAMASD resultant method has been integrated in an experimental version of the Zeus agent building toolkit. A simple constraint specification language was developed and implemented in the tool to represent design constraints. Furthermore, an algorithm for allocating roles to agents was developed.

The applicability of the RAMASD method has been tested by applying it in two business focused case studies using the extended Zeus agent-building toolkit. The case studies concerned providing support to mobile workforce and operating a B2B electronic marketplace respectively. The value of RAMASD in regards to reducing design complexity has been assessed by using a specially constructed evaluation framework and by comparing it in detail with Gaia, a representative ABS design method, in the context of the first case study. The evaluation results clearly demonstrate the superiority of RAMASD in all cases.

1.7 Case Study Descriptions

The original rationale for establishing the research project described in this thesis was the realisation that existing ABS design methods cannot deal with the degree of complexity inherent in designing ABSs to support BT's business and operational processes. The case studies used to evaluate the proposed approach are therefore representative of this type of systems.

The first case study concerns support of BT's mobile workforce. BT has about 25,000 mobile workers performing about 150,000 repair tasks everyday across the UK [121]. Supporting this workforce includes the following three dimensions considered in the first case study: a) travel management, b) teamwork coordination and c) work knowledge management. This case study was selected to demonstrate how RAMASD could cope with quantitative non-functional aspects and with design heuristics.

The second case study concerns COVISINT, a B2B electronic marketplace (B2B Exchange) concerning automotive industry. B2B electronic marketplaces offer a variety of services including business directories, auctions, supply-chain management and asset re-deployment and disposal. This case study was selected to demonstrate how RAMASD handles qualitative non-functional aspects and organisational settings.

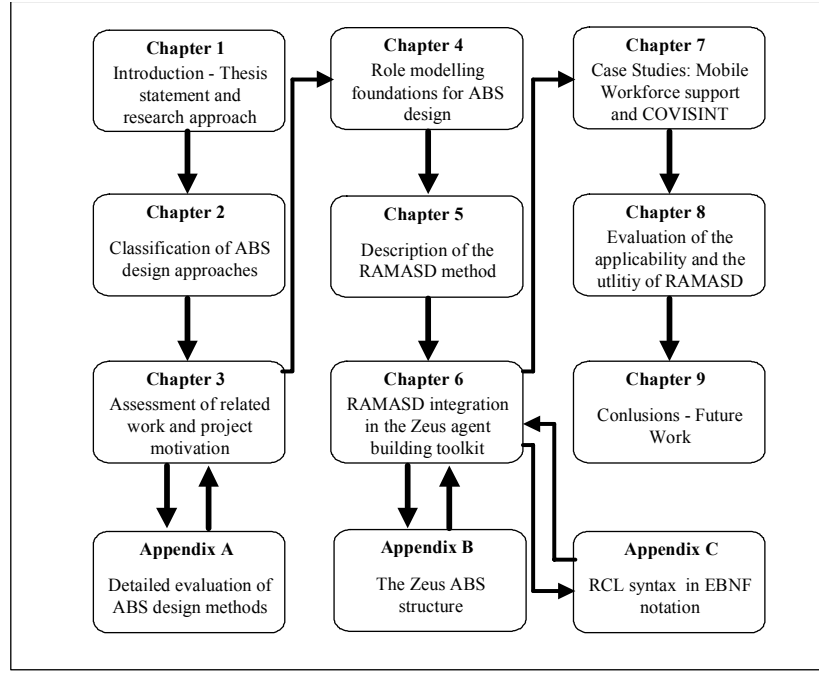


Figure 1.2: Thesis organisation

1.8 Thesis Organisation

The structure of the thesis is demonstrated in Figure 1.2. Chapter 2 provides a classification of current ABS engineering approaches with respect to the methods they include for the design phase. For each category in the classification scheme, a representative approach is examined in Chapter 3 using an evaluation framework concerning design complexity. The examination results illuminate on-going research challenges and provide the foundations for further study.

To this end, the foundations of the Role Algebraic Multi-Agent System Design (RAMASD) method are established in Chapter 4. The suitability of roles and role models as appropriate constructs to represent complex behaviour is described. This is done by providing details of how role modelling is used in software engineering and in social systems and in particular role theory. Furthermore, a number of approaches are compared to identify which role modelling aspects are suitable for ABS design. Finally, the case for formalising role relations and using those relations to drive allocation of roles to agents is made.

Chapter 5 elaborates on the expanded definition of roles introduced in Chapter 4 and presents RAMASD, a systematic method for designing ABSs using role modelling. RAMASD moves forward from the traditional definition of roles as conceptual modelling constructs and introduces a novel view of roles as representations of pragmatic behaviour including organisational knowledge and non-functional aspects. In addition, the primary innovation of RAMASD, a formal model of role relations termed role algebra, is discussed. The role algebra forms the basis for defining design constraints at a high abstraction level thus reducing design

complexity. Furthermore, the semi-automatic design process of RAMASD is described. Semi-automation is possible by having a clear separation of manual and automatic design steps based on the synthesis problem solving approach and it further reduces design complexity. Finally, Chapter 5 concludes with a discussion about how RAMASD can be incorporated within role-based ABS engineering methodologies.

Chapter 6 focuses on the RAMASD software prototype, built by extending the AgentGenerator tool of the Zeus agent building toolkit. It involves a graphical environment supporting editing, storing and instantiating roles and role models and automatically allocating roles to agent types by appropriate constraint problem solving algorithms. This chapter further presents RCL, a simple specification language for describing constraints on roles and role characteristics and a heuristic role allocation algorithm that can find a feasible design solution if one exists.

Chapter 7 presents the application of the RAMASD method in two case studies concerning mobile workforce support and COVISINT, a B2B electronic marketplace. For both case studies, the RAMASD steps from role modelling to role allocation and instantiation are described in detail. In each case study description emphasis is given to different RAMASD aspects. In the first case study description the focus is on quantitative non-functional aspects and on design heuristics, whilst in the second much attention is paid to qualitative non-functional aspects and organisational settings.

Chapter 8 provides an assessment of the value of RAMASD in regards to reducing design complexity. RAMASD is evaluated in two ways. Firstly, by comparing it with other methods with respect to design complexity using the evaluation framework introduced in Chapter 2. This is enabled by a discussion regarding how RAMASD has addressed all research challenges identified in Chapter 2 based on the case studies described in Chapter 7. Secondly, by performing a detailed comparison of RAMASD and Gaia in the context of the mobile workforce case study.

Chapter 9 discusses the originality and the contributions of this PhD. Furthermore, it concludes the thesis by suggesting and discussing areas for further work.

Appendix A provides a detailed assessment for each of the individual ABS design methods included in the comparisons of Chapter 3. A description of the structure of the ABSs produced by the Zeus agent building toolkit is provide in Appendix B. Finally, Appendix C describes the syntax of RCL, the role constraint language that is used to represent design constraints.

Chapter 2

Agent-Based System Design

This chapter classifies and reviews existing ABS engineering approaches focusing on the complexity encountered by designers whilst constructing realistic agent applications.

2.1 Designing ABSs

The agent paradigm has gained a wide popularity in the last decade and is generally considered to play a fundamental role in coping with the difficulties inherent in developing large-scale software systems [98], especially those that support flexible and evolving business organisations [99]. Many authors agree that ABSs can effectively provide the flexibility, adaptability and performance required from software supporting business operations [10, 19].

In the literature, a consensus regarding ABS engineering terminology, concepts and methodologies has hardly been reached yet [70, 146] and several open problems need to be solved. As established in Chapter 1, an important problem is how to support the design of large and complex ABSs operating in dynamic and open environments. This problem has to be addressed in order to be able to use ABSs in real world applications.

This chapter starts by defining some basic ABS concepts in Sections 2.2 and 2.3. In Section 2.4 it proposes a classification scheme of ABS engineering approaches, which is based on the design methods they include, and it investigates the needs and approaches of current ABS design methods. This paves the way for a detailed assessment of ABS design methods with respect to design complexity, which is discussed in Chapter 3. Finally, a summary of this chapter is provided in Section 2.5.

2.2 Overview of Agent Concepts

This section will gradually introduce a set of fundamental ABS engineering concepts, including *agent*, *agent architecture* and *agent roles*.

2.2.1 Agent-Oriented vs Object-Oriented Approaches

Agents are in many ways similar to software objects, for example both metaphors adhere to the principle of information hiding. However, a number of important differences exist, which make agents more suitable for building adaptable and intelligent software systems [97, 100, 212]:

- Objects are generally passive in nature and they need to be sent a message (method invocation) before they become active. Agents on the other hand can initiate some action.
- Although objects encapsulate state and behaviour implementation, they do not encapsulate autonomous behaviour to any extent. Thus, any object can invoke any publicly accessible method on any other object and the corresponding actions are performed. In contrast, to initiate a particular behaviour on an agent is to send it a message in a standardised communication language. The agent may or may not fulfil the request depending on its current state and goals that aims to achieve.
- Additionally, object-orientation fails to provide an adequate set of concepts and mechanisms for modelling dynamically changing, open systems. Individual objects represent dynamic behaviour at too fine granularity and method invocation is too primitive for describing the types of interactions that take place. Recognition of these facts led to the development of more powerful design patterns, application frameworks, and componentware. Agents, on the other hand, demonstrate goal oriented behaviour which is defined by particular goals and the perception of the environment. Hence, agent behaviour can change dynamically when the agent goals or the environment perceptions change.
- Finally, object-oriented approaches provide only minimal support for specifying and managing organizational relationships (basically relationships are defined by static inheritance hierarchies). Agents are defined along the lines of human behaviour. Hence, agent systems include complex organisational relationships among agents, which are similar to those found in human organisations.

Today's businesses have flexible structures formed dynamically and evolve to adapt to change and to open markets. The agent metaphor is suitable for software systems capable of meeting the requirements of today's business. This is because:

- Agents can adapt their relationships while the system is running. This matches business systems which tend to consist of a number of interacting components that dynamically come together, do business and then dissolve.
- Agents can adapt their behaviour based on their goals and on stimuli they sense from their environment. In contemporary business, each partner tries to maximize his benefit while cooperating with other partners. This requires flexible behaviour involving negotiation of business agreements using different strategies, and coordination based on changing rules and conventions. Agents can intuitively represent this behaviour by appropriate goals and negotiation and coordination protocols.

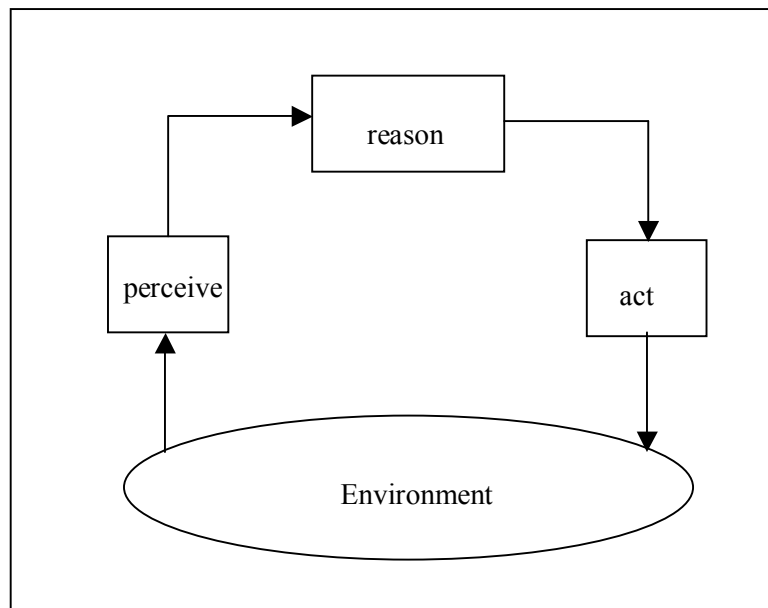


Figure 2.1: Perceive-Reason-Act cycle

- Agents can interact with other agents and legacy systems. Agents can support truly open business systems as they can communicate with other agents using standardised languages and act as communication front-ends to traditional software systems.

2.2.2 Defining the Term ‘Agent’

As argued in Section 2.2.1, agents are software components that exhibit a number of properties making them particularly suitable to cope with the dynamism and openness of contemporary software environments, such as the Internet.

There is currently an ongoing debate in the research community about exactly what constitutes an agent, which is far from reaching a consensus. In this thesis, a definition given by Wooldridge in [211] is adopted:

“An agent is an encapsulated computer system that is situated in some environment, and that is capable of flexible, autonomous action in that environment in order to meet its design objectives.”

This definition encapsulates a number of important points [96, 98]. Agents are:

- Clearly identifiable problem solving entities having particular objectives to achieve.
- Situated in a particular environment receiving input related to the state of that environment through their *sensors* and acting on that environment through their *effectors*.
- They are *autonomous*; they have control on both their internal state and their internal behaviour.

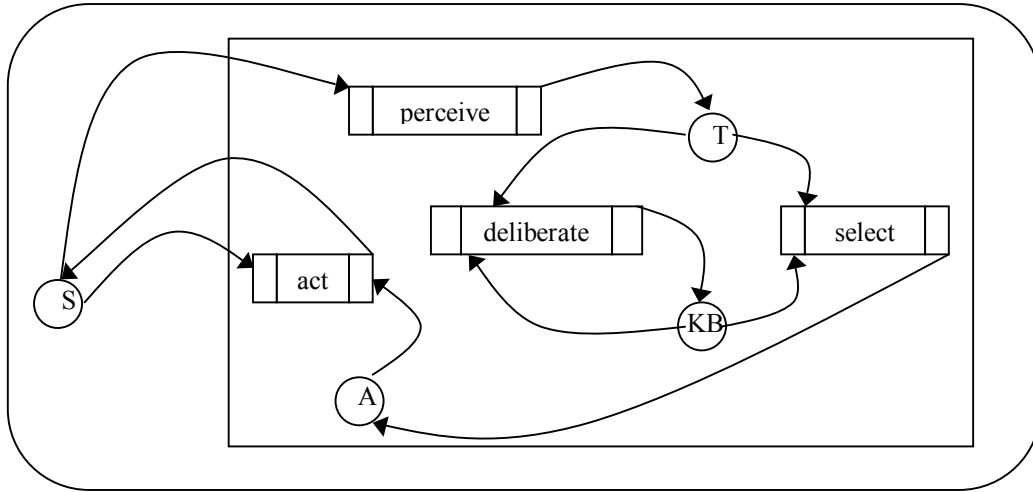


Figure 2.2: A simple agent formal model

- They are capable of exhibiting flexible (context-dependent) problem solving behaviour; they can be both *proactive* (take the initiative in order to satisfy their design objectives) and *reactive* (able to respond in a timely fashion to changes that occur in their environment).

The agent operation generally follows the *Perceive-Reason-Act (PRA)* cycle as depicted in

Figure 2.1. This cycle was originally introduced in [74] and was later used by other authors [122, 173, 206]. According to the *PRA* cycle, the agent receives some stimulus from the environment and processes this stimulus with its perceptual apparatus. Subsequently, the agent starts a reasoning process that combines the newly incorporated information and the agents existing knowledge and goals and this then determines possible actions of the agent. The best of these possible actions is then selected and executed. The action activation changes the state of the environment, which in turn generates new perceptions for the next cycle.

When the agent is purely a software system operating in a software environment then it is called a *software agent* [75, 146]. The concept of software agent was first considered within efforts to mitigate the compatibility problem among various types of heterogeneous legacy software components that had to communicate to exchange information [75]. The information exchange was standardised via the use of some common communication language and the software components that were able to communicate with it. In due course, several additional characteristics and capabilities were added to software agents including autonomy, mobility and sophisticated reasoning, which were, perhaps rather wishfully, called *intelligence*. A good overview of software agent concepts and a classification of different software agent types is given by Nwana in [146].

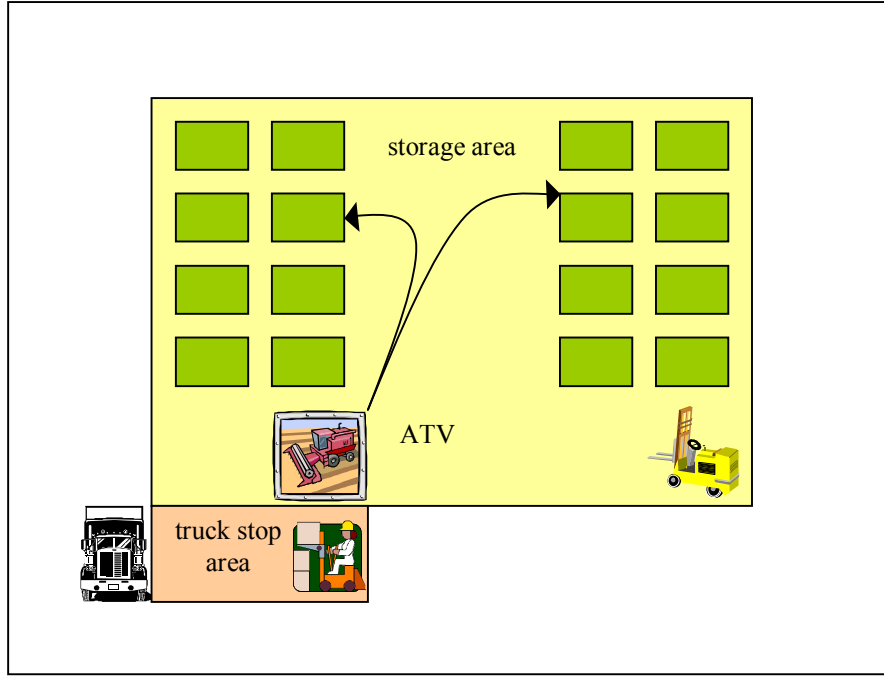


Figure 2.3: A container terminal yard agent

2.2.3 A Simple Agent Formal Model

To describe the *PRA* cycle more concisely and formally, a notation based to the one described in [74] can be used. The environment the agent is situated in can be represented by a unordered set S of states. An agent can be described as a 7-tuple $(KB, T, A, perceive, deliberate, select, act)$, where KB is a knowledge-base that contains the acquired knowledge of the agent, T is a set of partitions of the environment S which includes the possible perceptions of the agent and A is a set of possible actions of the agent. The agent behaviour can then be defined by four functions: The *perceive*: $S \rightarrow T$ function which determines how the state of the environment is perceived by the agent, i.e. it restricts the amount of information available to the partial information accessible by the agent. The *deliberate*: $KB \times T \rightarrow KB$ function updates the agent knowledge base after reasoning based on the newly received perceptions. The *select*: $KB \times T \rightarrow A$ function determines the best action for the current cycle and the *act*: $A \times S \rightarrow S$ function changes the state of the environment accordingly. In Figure 2.2, the agent formal model components and the information flow between them are depicted in a manner similar to the one introduced in [62].

2.2.4 An Example of a Simple Agent

To clarify the concepts described in Sections 2.2.2 and 2.2.3 a simple example from the area of agents in manufacturing [181] can be used. An agent guiding an automatic transfer vehicle (ATV) in a container terminal yard is considered. The ATV agent is responsible for unloading incoming containers from trucks and storing them on piles in the storage area. Figure 2.3 shows

such a facility with several container storage places, one ATV agent and a truck that has just delivered some containers that must be unloaded. Considering the simple agent model described in Section 2.2.3 above, the scenario is described as follows: The environment S of the ATV agent can be modelled by a grid world with labelled grid locations; the possible actions A of the ATV agent are *pick_container*, *transfer_to_location* and *drop_container*; and the robot's perception T is the content of the location right in front of the ATV agent and the knowledge base KB of the ATV agent contains the destination address of each container delivered by a truck.

The *Perceive-Reason-Act* cycle of the ATV agent starts when the *perceive* function determines the presence of newly arrived containers (assuming that the default waiting position of the ATV agent is at the track stop area). Subsequently, the *deliberate* function decides that the only possible action is *pick_container*. The *pick_container* action is subsequently scheduled for execution by the *select* function and executed by the *act* function of the ATV agent. As a result of this action, the state of the environment changes (because the ATV agent is now holding a container) and thus the next *PRS* cycle starts. In the next *PRS* cycle the ATV agent determines the destination of the container and based on that it transfers the container in the appropriate place in the storage area of the container terminal yard. The AGV agent stores the container in the appropriate place in a subsequent *PRS* cycle and its operation continues by returning to the track stop point.

2.2.5 Using Roles to Model Agent Behaviour

In the simple example described in Section 2.2.4, the problem solving capabilities that are necessary in the problem domain are directly associated with the agent. This approach, however, can be restrictive and impractical when the agent has to modify its capabilities to adapt to dynamically changing requirements or to use different capabilities in different circumstances [108, 209]. For example, if the ATV agent was to be used to unload, carry, and store other items as well, i.e. huge drain pipes, then all AGV functions would need to be modified explicitly. Therefore, it has been suggested that a modelling concept able to package multiple agent capabilities is required [51, 106]. An appropriate concept is the concept of *role*.

The role concept originated in sociology [15] and it is also used in organisational theory [72] and business process modelling [115] to represent positions and responsibilities in business organisations. When more than one role interacts within some context they constitute a *role model* [2, 108].

Several definitions of the role concept exist in the agent research community that differ mainly in what they consider as role properties. For example, Kendal in [108] defines role as a position and a set of characteristics including tasks, responsibilities, collaborators and planning,

coordination and negotiation capabilities. Weiss in [201] defines roles as "*the functional or social part which an agent, embedded in a multiagent environment, plays in a (joint) process like problem solving, planning or learning*". In all definitions, however, roles are modelling abstractions of some concrete behaviour. When the characteristics of a role are also characteristics of an agent, then it is said that the agent plays that role [108]. When designing agents using roles, the agent characteristics are determined by composing the characteristics of the individual roles the agent plays. The different types of role definitions available in the literature are discussed in detail in Chapter 4.

Formally speaking, the concept of a role can be modelled as an extension of an agent's current knowledge. The possible actions the agent can take and the *perceive*, *deliberate*, *select* and *act* functions. Thus, agents that can play a number of roles from a set of roles R are described by the 7-tuple: $(KB \cap KB_r, T, A \cap A_r, perceive \cap perceive_r, deliberate \cap deliberate_r, select \cap select_r, act \cap act_r)$ where $r \in R$.

To illustrate this definition, the example given in Section 2.2.4 is extended by adding a second role to the roles the AGV agent can play. Let us denote by "*carrier*" the role corresponding to the original AGV agent behaviour. If the possibility of the AGV searching for a container in the container terminal yard and informing a human operator accordingly is required, the AGV agent with the role "*verifier*" could be applied. The role *verifier* would include an appropriate $perceive_{verifier}$ perception function, which would make it possible to receive commands from a human operator and to determine status and position of an existing container already in the terminal yard.

In all areas where roles are used, a major problem is the delimitation of roles that occurs within the context of interest. Not every set of behavioural characteristics can be regarded as a role, there must exist some special properties that make such a set a role. This thesis proposes a method aiming to assist designers in this task. The proposed method is discussed in detail in Section 5.2.

2.2.6 Agent Architecture

The agent concepts discussed so far are useful to describe agent behaviour but they cannot be directly mapped to some executable software system. This section discusses about how these theoretical concepts can be mapped to executable software based on an intermediate layer of abstraction, which includes appropriate models that refine the abstract definition of an agent into a more concrete specification. The set of models in the intermediate layer of abstraction is called *agent architecture* [62, 83, 210]. Furthermore, the term *agent architectural specification* is used to refer to the resulting specification.

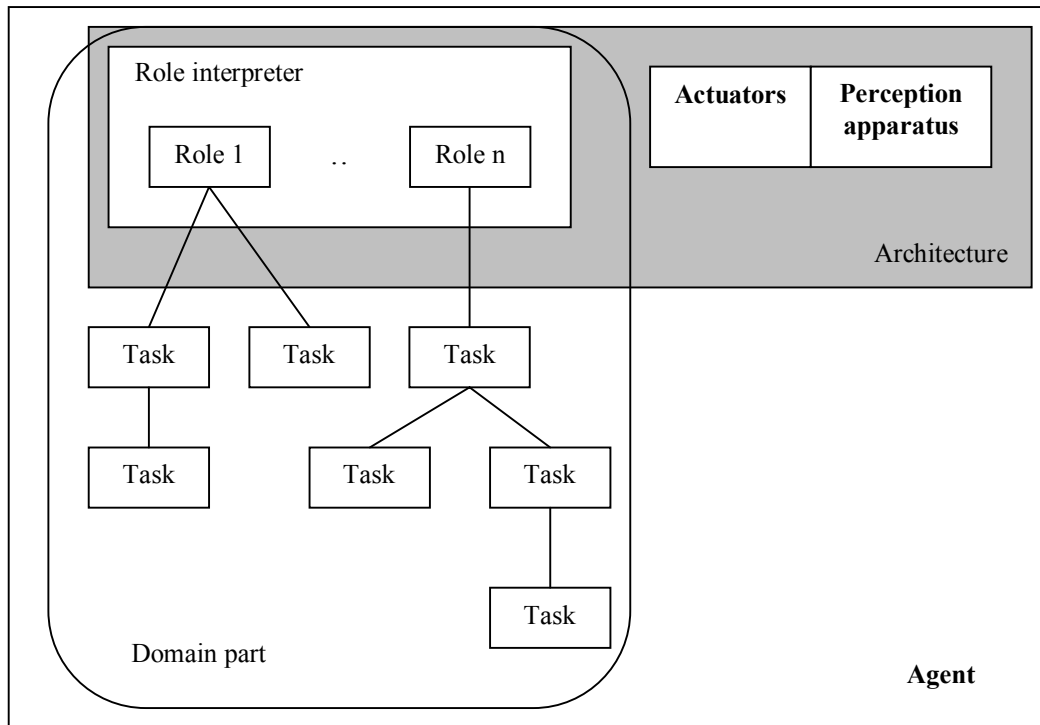


Figure 2.4: Agent internal components

When the agent is defined in terms of roles it plays, its architecture provides a runtime environment that is capable of executing the given roles. The above relation is represented graphically in Figure 2.4. The concept of an agent encloses the architecture that contains the perception and actuation subsystem as well as a role interpreter. The role interpreter links the domain-independent agent architecture to the domain specific aspects of the different roles by associating each role with a particular task tree.

The relation between the roles an agent plays and the architecture of the agent is complementary. The roles are the application functionality the agent has to deliver and the architecture of the agent is the means to deliver this functionality. For example, the agent architecture can be considered as a run-time environment for executing abstract agent definitions in a similar way that the Java Virtual Machine provides the means to execute Java code. However, not all agent architectures, for example the one proposed in [22], support the role concept.

In the example of the container terminal, the hardware of the AGV agent corresponds to the agent architecture that implements the runtime environment for the possible roles the AGV agent plays. The roles are modelled as task trees, e.g. the “*carrier*” role has the subtasks of checking for incoming containers, determining the destination of each container and then taking each container to the indicated destination.

Many agent architectures are based on cognitive models. One of the most prominent examples of such a cognitive model is the BDI model [162]. According to the BDI model an agent is described by its *Beliefs* that determine the current world knowledge of the agent, its *Desires* that determine the goals the agent needs to achieve and the *Intentions* that are generated from reasoning about the current beliefs and goals and therewith determine the best possible actions. A variation of the BDI model is used in the Zeus agent architecture [147], which is based on *Facts*, *Goals* and *Tasks*. The Zeus agent architecture is the one used in this thesis because it provides an environment for rapid development of agent applications which incorporates the concept of roles albeit only as an analysis concept.

Agent architectural specifications are still difficult to transfer to executable code. Therefore, further refinement is required. Wooldridge in [208] suggests three possible means to achieve this goal. The first possibility is to use functional refinement, which is common in most standard software engineering environments. The second one is direct execution of the specifications, which implies powerful description languages and runtime environments. The third possibility is compilation of the abstract architectural specification into executable code. In this thesis, the approach of compiling abstract agent architectural specifications to Java source code is followed. This approach has many advantages including portability, fast execution and direct interoperability with conventional software written in Java.

2.3 Agent-Based Systems

This section focuses on systems containing multiple agents and describes the main concepts involved.

2.3.1 Overview

Agents operate and exist in an environment, which typically is both computational and physical. The environment might be open or closed, and it might or might not contain other agents. Although there are cases where an agent can operate usefully alone, the level of today's interconnection and networking of computers require agents to interact with other agents in order to fulfil their objectives. In that case, it is more convenient to deal with those interacting agents collectively, as a society of agents [87] often referred to as *Multi-Agent System (MAS)* [201] or *Agent-Based System (ABS)* [181]. As established in Chapter 1, the term *Agent-Based System (ABS)* is adopted in this thesis and it is used to refer to a society of interacting agents.

There are numerous formal definitions of ABSs in the literature, e.g. [22, 62, 68]. Following [122], a simple formal model of an ABS based on a set structure can be used to describe the ABS concept:

$$\{S, (T, KB, A, perceive, deliberate, select, act)_i\}$$

where S denotes the environment just like in the agent formal model given in Section 2.2.3. Each agent is represented by the same 7-tuple as in Section 2.2.3, but in addition it is associated with a unique identifier i that distinguishes it from the other agents.

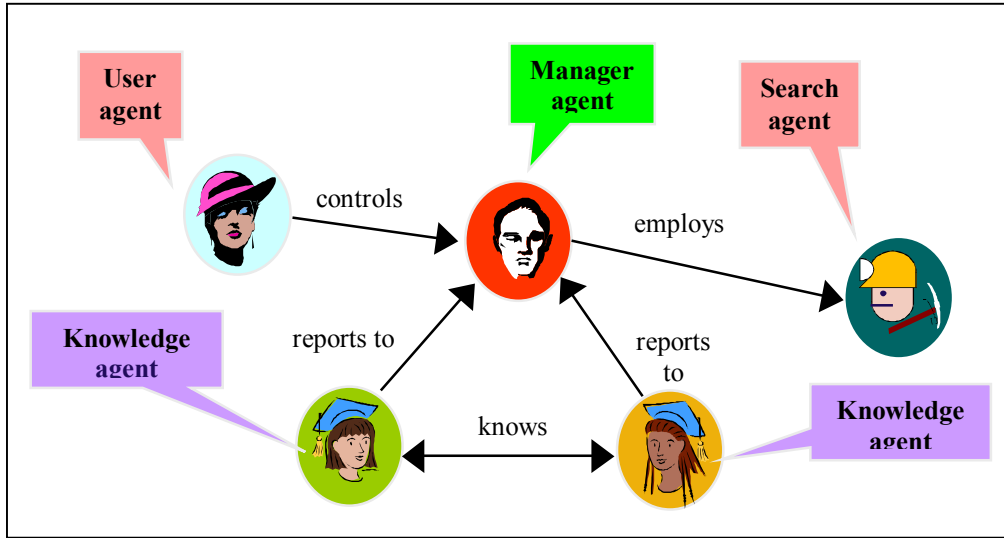


Figure 2.5: An agent organisation

2.3.2 Interaction in ABSs

The main feature of a system that is comprised of several intelligent entities is that a major part of the system’s functionality is not explicitly and globally specified, but that it emerges from the interaction between these individual entities [180]. Interaction is the mutual adaptation of agent behaviour while preserving individual constraints.

Interaction is not limited to explicit communication or to the case of message exchange. Weiss in [201] defines *agent interaction* as “any kind of agent behaviour that is related to other agents”. For example, ants may not explicitly communicate with other ants but still adapt their behaviour in a way that the entire ant society shows coordinated interaction. The interaction between ants can be carried out by several means including physical tactile behaviour, chemical substances, vision and others.

Coordinated interaction among several autonomous entities is the core concept of ABS design. The view in this thesis is that to leverage the desired ABS emergent behaviour, appropriate roles should be allocated to individual agents. To this end, the RAMASD method described in Chapter 5 is applicable.

2.3.3 ABSs as Organisations of Agents

Agent systems can be considered as organisations of autonomous, intelligent entities [62, 185, 209]. The organisational structure of an organisation determines how the entities within the

organisation relate to each other. There is no best organisation for all circumstances but instead organisation selection depends on a number of dynamically changing factors.

The issue of designing an agent organisation is related to sociology and to organisational theory. To design an agent organisation one needs to define the lines of the inter-agent communication, the individual agent functionality and the organisational authority relationships. An agent organisation does not exist for its own right; instead, it must have a purpose. The selection of the organisational relationships is done in a way to serve the overall purpose of the organisation. Agents in an agent organisation should be able to perceive the existence of other agents and to observe any organisational relationships that may exist between them. An example of an agent organisation is depicted in Figure 2.5. Each link in the figure has an associated characterization of its meaning that describes the nature of the connection between the organisational entities.

The criteria affecting an agent organisation design are numerous and highly dependent on several, possibly contradicting factors that may change dynamically [46, 185]. Therefore, finding an organisational structure that is suited for a particular functional specification and integrating with the application functionality are some of the most difficult parts of the ABS design process [62]. A technique that supports the developer in integrating organisational structures with application functionality is described in Chapter 5.

2.3.4 Software Complexity and ABS Design

The term complexity has been used in computer systems engineering with two different meanings [61, 207]: computational complexity and software engineering complexity. *Computational complexity* is primarily concerned with determining precise upper and lower bounds on the amount of computation time and memory space required to solve particular problems and on developing efficient algorithms for solving these problems. *Software engineering complexity* relates to how difficult it is to implement a particular computer system. In this thesis, the focus is on software engineering complexity and in particular on that inherent in ABS design.

2.3.4.1 Complexity in Software Engineering

The term complexity has been given many definitions in the literature and the majority of them are based on the Oxford English dictionary definition, referring to difficulty in understanding. It is considered that high software complexity results to low software quality [5, 25].

The difficulty in understanding has been the core of all definitions of complexity given in the context of software engineering. Software complexity refers to the “*difficulty of understanding and verifying software*” [92]. This difficulty, can be either described generically as “*the degree of comprehension of people that design software by putting together software components*” or

specifically in terms of software components — “a system property that depends on the relationships among elements and not an isolated element’s property” [26] — and other software characteristics — “an attribute of an object which is somehow associated with the following observables: number of its components or elements, kind or type of elements and structure of the relationships between elements” [45].

There is a consensus that lower software complexity provides advantages such as lower development and maintenance time and cost, less functional errors and increased reusability [26, 61, 226]. Therefore, it is common in the Software Metrics community to try to predict software qualities based on complexity metrics [61].

Software complexity can refer to software requirements specifications, to software design artefacts and to source code. Recently the focus has shifted to specification complexity since modern case tools can automate to a large degree the design and source code generation [130].

Many authors agree that there are multiple facets of software complexity [1, 26, 84, 207]. For example, Fenton and Pfleeger in [61] consider four types of software complexity: problem complexity, algorithmic complexity, structural complexity and cognitive complexity referring to the complexity of the underlying problem, the implementation algorithms and the structure of the implemented software and to the effort required to understand the implemented software. Hastings in [84] considers functional complexity, referring to the number of functions required to be developed, and problem complexity, referring to the difficulty in understanding the underlying problem. From the different types of software complexity, problem complexity and functional complexity are considered the most important [1, 17, 139, 186]. This view is followed in this thesis as well, as described in the next section.

2.3.4.2 Complexity in ABS Design

The sophisticated structure and properties of software agents increase the complexity inherent in ABS design [186]. For example, designing agents to operate in dynamic and open environments and carry out non-trivial tasks that require maximisation of some utility payoff function involves high software engineering complexity [207].

In this thesis, the interest is in the complexity that the ABS designer has to address when taking design decisions. As established in Section 1.2, this refers to the difficulty in understanding and manipulating the concepts and models needed for the detailed ABS design and in this thesis it is termed *design complexity*. Design complexity represents the combination of two particular facets of software complexity, functional complexity and problem complexity (see Section 2.3.4.1), which are considered worthwhile to try to reduce [1, 84].

The meaning of functional and problem complexity in the context of ABSs design is better illustrated with the following examples:

- *Functional Complexity*: A way to understand functional complexity is to consider the number of concepts that are needed to specify the software functionality at a particular level of abstraction [84, 128]. Along those lines, to understand functional complexity one can consider the number of concepts that are required to describe the collective behaviour of the ABS. In the example discussed in Section 2.2.5, the AGV agent was able to play two roles, namely “*carrier*” and “*verifier*”. If reasoning is carried out at that level of abstraction, in order to define the AGV agent, only these two roles are needed. If reasoning is done at the task level of abstraction, to define the AGV agent behaviour all the tasks it can carry out need to be explicitly specified. The latter case involves higher design complexity than the former.
- *Problem Complexity*: It is common in software metrics research to measure the complexity of the problem the software is solving by the number of invariants required for the problem specification [61, 84]. Along these lines, to understand conceptual complexity the number of specification constraints that are required to fully specify the behaviour of an ABS can be considered. For example, let us assume an ABS that supports the various administration procedures at an educational institution. Among other things, the university members need to access the library database to manipulate their library loans. If role modelling is used, this behaviour could be represented by the role *Library_User*. However, only members of that institution can use the library and the institution membership behaviour can be modelled by the *Institution_Member* role. Hence, an agent must be able to play both *Library_User* and *Institution_Member* roles in order to be conceptually consistent. If reasoning is carried out at that level of abstraction, only this role constraint that must characterise the agent behaviour needs to be specified. If reasoning is done on the tasks each agent can carry out, then obviously more constraints are required.

The above examples show that ABS design complexity depends on the level of abstraction the designer is working at. They also indicate that design complexity depends on automating some parts of the design process. For example, appropriate mechanisms to automatically combine the characteristics of the “*carrier*” and “*verifier*” roles when designing the AGV agent would relieve the designer from having to consider the details of these two roles and carry out the design manually. Enabling the designer to work at different levels of abstraction and semi-automating the design process is the basis of the proposed method to reduce ABS design complexity, which is described in Chapter 5.

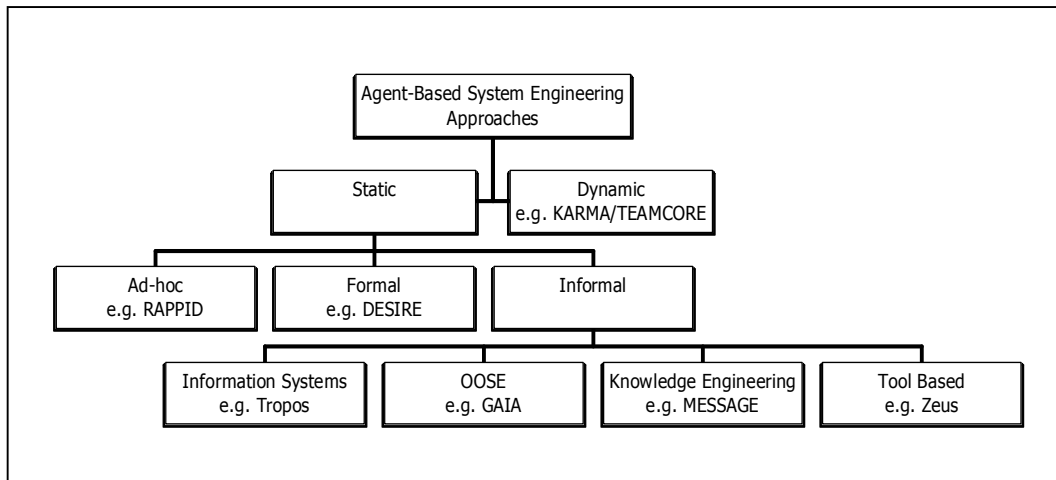


Figure 2.6: Classification of ABS engineering approaches

2.4 Classification of ABS Engineering Approaches

There are currently many different types of ABS engineering approaches ranging from simple strategies to comprehensive methodologies [90, 98, 154, 181]. As described in Section 1.2, this thesis is primarily concerned with the design of ABSs, namely with specifying the behaviour of the different types of agents and with deciding on the number of agents of each type that will be included in the system. The majority of existing ABS engineering approaches involve methods to support other software engineering phases as well, for example, requirements capture, analysis, code generation and testing. A systematic classification of ABS engineering approaches, with respect to the ABS design phase, is useful for better understanding the advantages and disadvantages of each approach,

Extending the classification introduced by Wooldridge in [208], a classification scheme for ABS engineering approaches is proposed and summarised in Figure 2.6. The criteria for the classification are: whether the design methods are applied before or after the deployment of the ABS; the degree of formality present in each approach; and the relevance of each approach with traditional software engineering methodologies. Each criterion corresponds to a different level in the classification tree described in Figure 2.6.

In terms of when design methods are applied, ABS engineering approaches can be classified as *static* or *dynamic*. In static approaches, design methods are applied only once before the deployment of the ABS. In dynamic approaches, design methods are applied on run-time, many times or continuously, resulting to reorganisation of the ABS.

Static approaches can be further classified as *ad-hoc*, *formal* or *informal*. The criteria for this classification are based on the techniques followed to specify the behaviour of each agent in the ABS. Ad-hoc ABS design refers to constructing ABSs without applying a systematic design

method. Formal ABS design concerns the use of formal methods for specification and verification of the agent behaviour. Finally, informal approaches include structured and systematic methods for designing ABS where the design decisions are taken based on heuristic rules and guidelines rather than rigorous criteria.

The majority of informal ABS design approaches originated and are closely related to traditional software engineering methodologies. Based on this relation, informal approaches can be further divided into four categories: those originating from *object-oriented software engineering (OOSE) methodologies*, e.g. Agent/UML [12], Gaia [209] and MESSAGE/UML [30], those that are extensions of *knowledge engineering methodologies* e.g. MAS-CommonCADs [91], those that are based on *information systems methodologies* e.g. TROPOS [23], those that are highly coupled with specific *ABS building toolkits*, e.g. Zeus [147] and those that have been developed for *specific industrial application domains* e.g. RAPPID [152].

The first category can be further broken down into approaches that focus on adapting existing software engineering standards and notations to ABSs engineering e.g. Agent/UML [12], approaches that aim to combine agent theories with existing software engineering methodologies e.g. GAIA [209] and those that combine elements from all approaches in a comprehensive way e.g. MESSAGE/UML [30].

2.4.1 Ad-hoc Approaches

The philosophy underlying ad-hoc ABS engineering approaches is that design decisions should be simple and clearly justified by the strengths offered by agent technology, for example autonomy and flexibility. Therefore, design in ad-hoc approaches is based on generic guidelines, which are followed in order to argue that the requirements of the application domain are better supported by utilising properties of the agent metaphor. For example, one such guideline is that each user should be paired with a software agent that will be able to act on his/her behalf to a certain extent. An agent is the most suitable software component for this responsibility as it is able to learn from past interactions and adapt to changing requests from its owner. An example of ad-hoc ABS design approach is RAPPID, which is discussed in more detail in Appendix A.1.

2.4.2 Formal Approaches

Formal ABS engineering approaches originated to improve the poor adoption of agent technology in industrial applications, which was caused by the lack of rigour of ad-hoc approaches [22]. They are mainly used in three ways [208]: for the *specification*, *systematic implementation* and *verification* of ABSs. Specification is used for the design of agent behaviour. In order to formally specify the behaviour of an agent, a theory describing the internal parts of an agent as well as how those parts interact to generate the agent behaviour is

required. Such a theory is called an *Agent Theory* [210]. A very common approach to formal agent theories is to use some temporal modal logic, namely a logic describing possible agent interactions over time. Two of the best known logical frameworks are the Cohen-Levesque *Theory of Intention* [37] and Rao-Georgeff *Belief-Desire-Intention* model [162]. The Cohen-Levesque model is based on two agent attributes: beliefs and goals. Other attributes, for example the notion of intention, are built from those. In contrast, the Rao-Georgeff model takes intentions as primitives, in addition to beliefs and goals.

A specification expressed in such logics describes the desirable behaviour of a system. For example, for two agents *a* and *b* aimed to support a manufacturing process control system, a specification formula might be [208]:

```

if
  a believes that valve 32 is open
then
  a should intend that b should believe that valve
  32 is open

```

A specification of the whole ABS might be constructed using such formulae to define the intended behaviour of the system. Specification is the starting point of every formal ABS design approach [22, 85].

Formal approaches to ABS design are often based on unrealistic assumptions, for example the *possible worlds assumption* [208], which impede the efficient mapping of specifications to appropriate implementations. A formal approach that does not suffer from this weakness, DESIRE [22], is discussed in more detail in Appendix A.2.

2.4.3 Informal Approaches

Informal ABS engineering approaches can be based on various concepts aiming to capture the domain knowledge and describe the agent system behaviour. Concepts related to different parts of the systems can be organised in different models. For example, an interaction model can describe interactions that take place between agents and an organisational model can describe the organisational structure of the actors in the business organisation. Such models can be used for both the analysis and design stages in ABS engineering. The models involved in informal approaches are based on different views each examining the ABS from a different perspective.

Informal approaches are often based on existing software engineering methodologies, which they extend to suit the particular requirements of agent-based applications. In particular, informal approaches are often based on methodologies from object-oriented software engineering, from knowledge engineering and from information systems. Furthermore, they may be tailored to specific ABS building toolkits. A common characteristic in informal

approaches is that the design of ABSs cannot be automated to any extent and is carried out manually by the designers based on informal guidelines and their experience.

2.4.3.1 Approaches Based on Object-Oriented Software Engineering

ABS engineering approaches that are based on object-oriented methodologies generally start from the full set of concepts and properties of the agent metaphor and try to adapt aspects from traditional object-oriented methodologies as required, for example Gaia [209]. Furthermore, many approaches extend object-oriented standards and notations to be applicable to ABS design. For example, Agent/UML [12] and MASE [205] extend the UML notation considering agents as specialisation of objects. As a representative example of such approaches Gaia is reviewed in Appendix A.3.

2.4.3.2 Approaches Based on Information Systems Engineering

Many ABS engineering approaches originate from the area of information systems engineering, for example [24, 56, 200]. Those approaches are based on the premise that new, open and evolving business models in areas such as e-Business [10] and e-Services [53] call for software systems which have open, evolving architectures that operate robustly and exploit resources available in their environment.

The main argument in those approaches is that the semantics of contemporary business transactions can only be captured if the specific business actors associated with the involved events and actions are explicitly represented in the information system in addition to passive business objects [200]. Therefore, to capture the dynamic aspects of information systems, such as the events and actions related to the ongoing business processes of an enterprise, it is necessary to make an ontological distinction between active and passive entities, that is between *agents* and *objects*.

The general philosophy of approaches to ABS engineering that originate from information systems methodologies is that they acknowledge the need to model dynamically evolving parts of the system using agents and agent relationships. For this purpose, they introduce expressive modelling mechanisms and they propose techniques for transforming agent-based conceptual models to traditional software engineering models. For example, Wagner in his Agent-Object-Relationship (AOR) approach [200] extends the Entity-Relationship approach to model dynamic system aspects using agents and relations between agents in addition to static entities. Wagner also proposes methods to transform agent conceptual models to relational, implementation-ready information system designs.

Information systems-based approaches claim to supersede other ABS engineering approaches because they are tailored to software systems that will operate in an organisational context. Such

methodologies aim to use the same concepts to describe the organisational environment within which the software system will eventually operate, as well as the system itself [31]. For example, in Tropos [31] the software system is represented as one or more actors, which contribute to the fulfilment of the stakeholder goals. Tropos is reviewed in more detail in Appendix A.4.

2.4.3.3 Approaches Based on Knowledge Engineering

Knowledge engineering methodologies are themselves considered suitable for modelling ABSs because of the knowledge intensive nature of agents. Therefore, they can conveniently provide techniques for modelling the agent knowledge and cognitive behaviour [90]. In addition, existing knowledge engineering tools, ontology libraries and problem solving method libraries can be reused. However, knowledge engineering approaches consider a knowledge-based system as a centralised one. Thus, they are not geared to the distributed or social aspects of the agents, or their reflective and goal-oriented attitudes.

Therefore, a number of ABS engineering approaches extend knowledge engineering methodologies, e.g. MESSAGE/UML [30], ComMoMAS [78] and MAS-CommonKADS [91]. The majority of them are based on the CommonKADS knowledge engineering methodology [176]. CommonKADS defines the modelling activity as the building of a number of separate models that capture salient features of the system and its environment. This is the case for the ABS engineering approaches based on CommonKADS as well. For example, MAS-CommonKADS includes six analysis models and three design models. A common deficiency in those approaches is that they do not describe the links between different specification models and the implementation of the ABS.

The most recent of this class of approaches to ABS engineering is MESSAGE/UML, which is reviewed and evaluated in Appendix A.5.

2.4.3.4 Tool-Based Approaches

A large number of commercial/research agent development toolkits have proliferated, the majority of them being in the public domain [165]. Agent development toolkits aim at facilitating the engineering of ABSs by rapid prototyping. The rapid prototyping approach they support is tailored to the specific implementation and underlying model assumptions they make. For example, in Zeus [147] an agent knows about particular facts, can carry out specific tasks and can have goals, while in Voyager [148] only reactive agents are supported. Another example is that in Jade [13] a full agent lifecycle including creation and removal can be specified. In the current version of Zeus there is not such provision and agents are assumed to execute infinitely. To illustrate and assess this class of ABS engineering approaches the Zeus ABS engineering approach is reviewed and evaluated in Appendix A.6.

2.4.4 Dynamic Approaches

Dynamic ABS engineering consists of having agents changing their organisational relations and their behaviour based on the stimuli of the environment and the changes in application requirements. The approach of having an ABS changing its structure and functionality on run-time is called *Self-Organisation* [76, 93, 136, 185]. There are many approaches to self-organisation ranging from those inspired by biological and chemical systems [156] to approaches based on heuristics and optimisation of mathematical functions [76] [79].

Reorganisation can be based on various primitive actions taken when appropriate reorganisation criteria are satisfied. For example, the *Organisation Self-Design* (OSD) framework, proposed by Ishida, Gasser and Yokoo in [93], includes the reorganisation primitives of composition and decomposition. Decomposition involves division of an agent into two similar agents while composition merges two agents into one. Reorganisation acts in OSD depend on a set of heuristic rules that can dynamically change the agent relationships, the agent knowledge, the size of the agent population and the resources allocated to each agent. Decomposition is performed as a result of environmental demands that are far too great for the existing agent organisation to handle. Composition may be invoked when communication overheads are potentially too high, or resource access response times too long to be tolerated. Therefore, the number of agents is reduced to free resources by limiting resource requests. The initial organisation starts with one agent containing all domain and organisational knowledge.

Other, reorganisation approaches do not involve only composition and decomposition. For example, Patisson et. al. in [159] address the organisational reconfiguration problem. Their approach focuses on repairing broken organisations by reallocating roles and responsibilities to new organisational nodes, when the nodes previously responsible for particular tasks are unable to perform them effectively. Further approaches to run time reorganisation are based on bottleneck analysis [76] and on arranging agents in hierarchical groups that are dynamically formed based on the agent capabilities and the application requirements [136].

To illustrate and assess this type of ABS engineering approaches the KARMA/TEAMCORE approach [192] is reviewed in Appendix A.7.

2.4.5 Overall Assessment

In all perspectives, it is clear that ad-hoc approaches provide the weakest support to ABS designers. The only advantage of ad-hoc approaches is their alignment to particular application domains, which can facilitate capturing application requirements. However, they do not provide any systematic support for the design stage and therefore they are difficult to use and error-prone.

Formal approaches provide rigorous support for verifying and specifying the design of ABSs and rigorous specification can be the basis for automating to some extent the design and even the implementation process, as is the case in KARMA/TEAMCORE [192] and Concurrent METATEM [68] respectively. However, the general approach of automatic synthesis of detailed agent specifications, although theoretically appealing, is limited in a number of important aspects [98]. Firstly, as the agent specification language becomes more expressive, for example a language based on first-order calculus, then the synthesis problem is harder to solve [208] and there is no algorithm guaranteed to find a solution. Secondly, when the language is based on first-order logic the algorithm complexity of theorem-proving can be exponential and hence not practical for real world ABS design. Thirdly, formal techniques based on mathematical theories are usually difficult to apply for the average software engineer [208] and do not facilitate communication with customers who do not have a formal mathematical background. Therefore, many authors argue that at least some steps in methodologies for ABSs engineering should be left informal [98, 160].

Informal approaches require the designer to address most of the system complexity based on creativity and intuition alone. This can be a serious problem in engineering large, real world ABSs. Furthermore, informal approaches lack a semantic framework and notation that would allow any verification of the design decisions. This complete lack of formality may result in error-prone designs and it does not allow any automation of the design process. This contrasts with the view supported by many authors in ABS design [186, 193] and software design in general [124], that to reduce development effort the design process must be automated to a certain extent.

Dynamic approaches offer the advantage of being able to adapt the ABS to dynamically changing requirements, which is of great significance considering the open and dynamic environments that ABSs need to operate in. However, dynamic reorganisation approaches suffer from a number of problems. Firstly, they consume a lot of run-time system resources as the agents in the ABS need to communicate frequently in order to carry out reorganisation acts. The assessment of the impact of the reorganisation in system resource consumption is a subject of ongoing research [76, 185, 192]. Secondly, the reorganisation signals may take a long time to propagate and hence the system behaviour is not always clear. Thirdly, the system performance may deteriorate for the same reason [119]. Therefore, it is concluded that dynamic reorganisation should be reduced as much as possible and it should be used only when necessary, for example when the agent environment changes due to the inherent openness of the agent systems.

The above analysis leads to the following conclusions regarding the desirable characteristics of ABS engineering approaches:

- They should be primarily static and involve only minimal reorganisation.
- They should be informal so that they can be easier to use by the average software engineer.
- They should have sufficient formal underpinnings so that some routine steps of the design process could be automated.

2.5 Summary

This chapter provided an overview of the basic terms characterising the notions of agent and ABS. It defined the main terms and concepts used in the ABSs design field and investigated its needs and approaches.

To provide the necessary background for discussing the main thesis topic the basic ideas underlying intelligent agents and ABSs have been outlined. Starting from a very general formalisation of an agent, the concepts of agent role and agent architecture were discussed. Moving onto systems with several agents, the fundamental aspects, such as interaction and the social dimension of an agent society, were described.

To better study the strengths and weaknesses of current ABS engineering approaches, with respect to ABS design, a classification scheme was introduced. Existing approaches can be classified as static or dynamic. Static approaches can be classified as ad-hoc, formal or informal. Furthermore, informal approaches can be classified as originating from object-oriented programming, knowledge engineering, information systems or from specific agent building tools. Dynamic approaches are time and resource consuming, formal design approaches are difficult to apply in practice and to produce implementations of ABSs and ad-hoc and informal approaches may result in inconsistent and performance problematic designs. The analysis conducted using the classification scheme led to the conclusion that an ABS engineering approach should be primarily static to involve minimal reorganisation, informal to be easy to understand by the average designer and have sufficient formal underpinnings so that part of the design process could be automated.

Chapter 3

Assessment of ABS Engineering Approaches

This chapter evaluates a representative selection of ABS engineering approaches against a number of design complexity related criteria and the results are used to motivate and guide further work in the area.

3.1 An Evaluation Framework for ABS Design

The classification of ABS engineering approaches has shown a large diversity of approaches with a variety of objectives and different levels of design complexity. To evaluate this complexity, and outline areas for improvement, a comprehensive evaluation framework is proposed here. The analysis is naturally focused on the way ABS design is done.

The proposed framework was inspired by attempts to understand and discuss the issues involved in ABS design approaches in a systematic manner [90] and it is based on similar frameworks for understanding and evaluating object-oriented software engineering approaches [195] and on approaches to comparing ABS toolkits [55, 169, 182] and measuring software complexity [61]. The issues included in the framework have been selected based on their relevance to the hypothesis that has been made in Section 1.5.

The framework examines ABS engineering approaches from different views, *Concepts*, *Models*, *Process* and *Pragmatics*, which are summarised in Figure 3.1. This idea is an analogy to the fact that there may be different abstractions from the same reality [122]. Different views describe an ABS engineering approach from different perspectives. Each view represents a set of conceptually linked aspects. For example, the implementation language and the use of standard notations are both related to the implementation and hence they should be considered aspects of an implementation-related view.

As mentioned in Section 2.3.4, two important facets of design complexity are those related with the concepts involved and the functions required to design the software. This gave rise to the Concepts and Models views in the proposed evaluation framework. Furthermore, design complexity should also be assessed with respect to the process followed to develop the software [61]. Therefore, the Process view was considered as well. Finally, the Pragmatic view was considered since the aim of the framework was to assess design complexity of ABS design methods in the context of real-world agent applications.

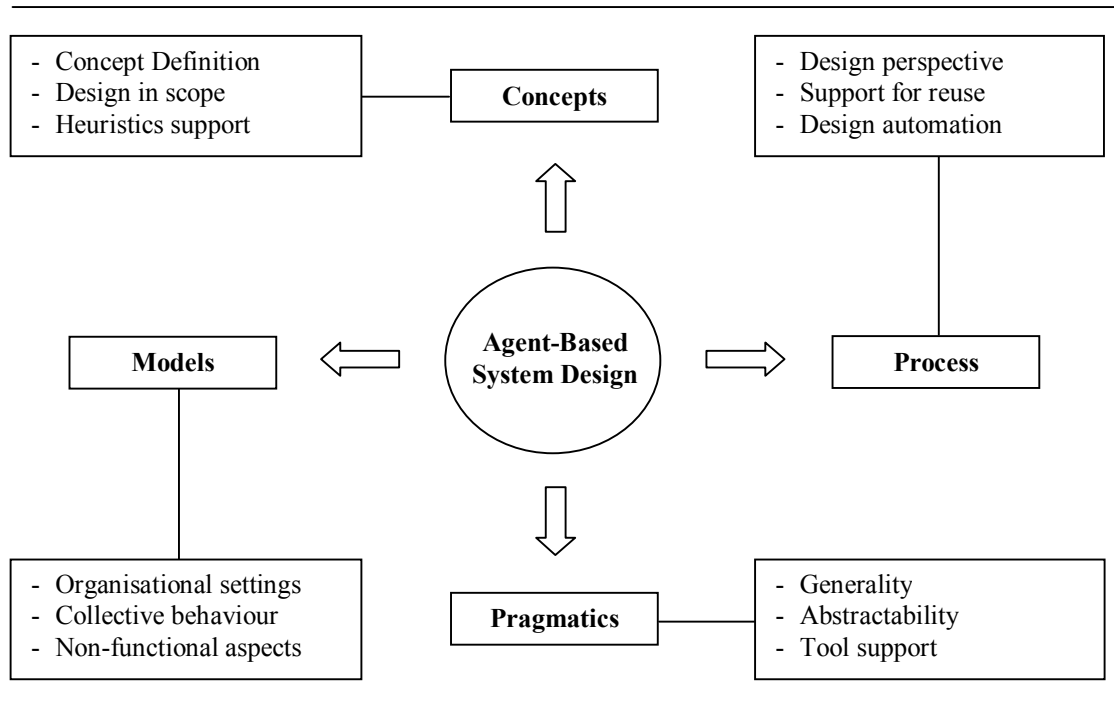


Figure 3.1: A framework for comparing ABS engineering approaches with respect to design

The aspects of each view were selected based on known issues of concern in ABS design which are described in more detail in the following sections. Particular attention has been given to aspects relevant to the ABS design issues of interest identified in Section 1.3: non-functional aspects, design heuristics and reusing design knowledge.

When assessing an ABS engineering approach using the proposed framework, a ranking scheme for each aspect is applied. The ranking is based on subjective, qualitative values, for example, low, medium, high. The possible ranking values are discussed together with the different aspects of the framework in Sections 3.1.1-3.1.4. A summary of the ranking values together with a short explanation of the framework aspects is given in Table 3.1.

3.1.1 Concepts

The concepts view concentrates on which modelling concepts are used in each approach to model and represent the ABS. In this view, the following aspects are of interest:

1. *Concept Definition*: This aspect refers to restrictive premises concerning the *agent architecture* and the *type*¹ of agents that can be designed. Based on this criterion, an ABS engineering approach can be characterised as *open*, *bounded* or *limited (highly bounded)*. An approach is open if it does not assume a particular agent architecture and does not

¹ An agent type is a class of agents with similar capabilities and purpose.

produce specific agent types, for example Gaia [209]. Alternatively, an approach may be bounded to a particular agent architecture, as is the case with Tropos [77], which assumes only BDI agents. Furthermore, an approach may be limited to producing only specific types of agents, for example RAPPID [158], which considers only two types of agents: *Component Agents* that represent humans and *Characteristic Agents* that represent parts of a product design system. It is preferable for an approach to be open as it can produce more types of agents and ABSs.

2. *Design in Scope*: This aspect refers to whether an approach includes specific methods and guidelines for the design phase of the ABS engineering lifecycle. For example, MESSAGE/UML [30] covers only the analysis phase of the engineering lifecycle while MASE [205] covers analysis, design and also part of the implementation. As far as it concerns explicitly supporting the design phase of the ABS engineering lifecycle, an approach can be characterised as *true* or *false*.
3. *Heuristics support*: This aspect refers to the formal support for applying heuristic guidelines and tips when designing the ABS. Heuristics can be either specific to ABSs design, for example the *sphere of responsibility* heuristic is specific to role-based approaches for ABS design like Zeus [38], or they can be generic. For example, there is a consensus in the software engineering community that designed components should have low *coupling* and high *cohesion* [179]. The formal support for design heuristics provided by an approach can be characterised as *true* or *false*. In the case of existing formal heuristics support, the approach provides formal techniques that can be used to ensure application of the design heuristics. For example, in KARMA [192] heuristics can be specified as constraints in the STEAM specification language. In contrast, in RAPPID [158] there is no rigorous way for ensuring that design heuristics have been applied. It is preferable for an approach to provide formal heuristics support since this increases the quality of the ABS designs.

3.1.2 Models

The *Models* view refers to the models that are used to represent different parts of the ABS or issues of particular interest and the techniques that are used to create and manipulate those models. This *Models* view includes the following aspects of interest:

1. *Organisational settings*: This aspect refers to how organisational settings are represented in each approach and whether they can be considered as first-class design constructs. Organisational settings may be represented by explicit models. For example, in Zeus [147] they are represented by role models, or they may be implied by the agent functionality, for example in DESIRE [22] and MASE [205]. The support for explicit

modelling of organisational settings by an approach can be characterised as *true* or *false*. Organisational settings need to be considered as first class design constructs [64, 220] and therefore ABS engineering approaches should support explicit modelling of organisational settings and provide the means to use them directly in designing ABSs.

2. *Collective Behaviours*: This aspect refers to whether an approach includes appropriate first-class modelling constructs to represent and reason with collective behaviour resulting from agent interactions. Collective behaviour may be implicitly modelled via the individual agent behaviour, as is the case in RAPPID [158], or it can be explicitly modelled by appropriate models; for example, in Zeus it is modelled by role models [147]. Many authors argue that collective behaviour as well as social and organisational abstractions should be considered as first class design constructs enabling the agent system designer to reason at a high abstraction level [108, 150]. The support for explicit modelling of collective behaviour by an approach can be characterised as *true* or *false*. It is preferable for an approach to support explicit modelling of collective behaviour because the abstraction achieved reduces development effort and specification errors [129].
3. *Non-functional aspects*: This aspect refers to the way that non-functional aspects are considered in each approach. Non-functional aspects can be implicitly modelled within individual agent behaviour — for example, in Gaia [209] — or can be explicitly represented by appropriate modelling constructs, for example, in Tropos [77]. Furthermore, it is possible for non-functional aspects to be taken into account by adjusting the agent behaviour on run-time [76, 185]. The support for explicit modelling of non-functional aspects by an approach can be characterised as *true* or *false*. It is preferable for an approach to support explicit modelling of non-functional aspects and they should be considered as early in the engineering process as possible [35].

3.1.3 Process

The process view concentrates on the steps that are executed in order to construct the models discussed in the *Models* view and on techniques that support and assess those steps. In particular, this view is concerned with the following aspects:

1. *Design Perspective*: This aspect refers to the perspective from which each approach views the ABS design. The perspective can be *top-down* or *bottom-up* or *both* (top-down and bottom-up) depending on how the design of the ABS progresses. In the top-down perspective, the design models are constructed by refining high-level models of the agent organisation, such as in Gaia [209]. In the bottom-up perspective, design models are progressively composed from existing finer-grain models thus supporting

reuse, for example in [111]. ABS engineering approaches should support both, such as MESSAGE/UML [30].

2. *Support for Reuse*: Reuse refers to whether the approach supports using previous knowledge in designing an ABS. Approaches that explicitly address reuse provide steps, deliverables and heuristics for the identification, construction, testing, demonstration and application of reusable components. For example, in the Zeus toolkit methodology [147] there are guidelines for creating, storing and reusing negotiation strategies when specifying agent interactions, whilst in RAPPID [158] there are not such facilities. Regarding supporting reuse, an approach can be characterised as *true* or *false*. Systematically supporting reuse is highly preferable since it reduces development effort [141].
3. *Design Automation*: This aspect refers to the degree of formality that exists in the specification models of the approach. The higher the degree of formality the more the design process can be automated [124]. Some process steps should definitely be carried out based on the judgement of the human designers. For example, the selection of roles in the analysis phase in Gaia has definitely to be carried out by the human designers [209]. However, some other process steps could be automated and carried out by a software tool. For example, it is possible to automatically create agent system designs from the analysis models using formal graph transformations [50, 52, 186]. The degree to which the process steps of an approach can be automated can be characterised as *high*, *medium* or *low*. For example, the DESIRE [22] approach can be highly automated, as many steps are formally defined using mathematical techniques, while RAPPID [158] cannot be automated since the design process is not formally defined to any degree. It is preferable for an approach to be highly automated as this reduces development effort and development errors [1].

3.1.4 Pragmatics

This view focuses on the pragmatics of each ABS engineering approach. In other words, this view refers to how practical the approach is for the design of real-world agent systems. The aspects of interest in the pragmatics view are the generality, the complexity handling and the tool support involved in an approach.

1. *Generality*: The generality of each approach refers to what development context the method is appropriate for. Generality has to do with restrictive premises that affect the applicability of the approach as far as it concerns the environment and the application domain. The generality of an approach can be characterised as *high*, *medium* or *low*. High generality means that the approach can be applied without any significant

restrictions, e.g. Tropos [77]. The generality is medium when there are considerable restrictions but the applicability of the approach is still wide. For example, Gaia [209] assumes a closed ABS and a small number of cooperating agents. In contrast, RAPPID [152] is limited in the sense that it can be applied to design ABSs that will only be used to support industrial product design and, therefore, its generality is low.

2. *Abstractability*: This aspect refers to whether there is formal support to handle the design complexity inherent in an ABS engineering approach. As mentioned in Section 2.3.4.1, design complexity refers to how difficult it is to understand the required concepts and techniques involved in an approach and apply them to design an ABS. As high design complexity results in error-prone software products [129] and increased development effort [25], it should be taken into account when selecting an ABS engineering approach.

One of the main factors affecting ABS design complexity is whether designers are allowed to work at different levels of abstraction. For example, the complexity involved in Tropos [24] is higher than the complexity involved in DESIRE [22] since in the latter it is possible to specify agent behaviour at different abstraction levels which are formally described. Approaches to handling design complexity by means of working at different abstraction levels can be characterised as *true* or *false*. This generic criterion is used to characterise the complexity handling of ABS engineering approaches in Section 3.2.

3. *Tool support*: This aspect is concerned with whether there are tools supporting the realisation of the approach. For example, the role-based approach introduced in [147] is supported by the Zeus agent building toolkit, which assists the users in designing ABSs. On the other hand, there is no tool support for the Gaia approach [209] and the engineer is responsible for manually creating all the relevant models. The tool support of an approach can be characterised as *true* or *false*. It is preferable for an approach to be supported by CASE tools since this greatly reduces development effort and development errors [129] and increases the usability of the approach since it automates mundane and repetitive tasks [143].

It must be noted that some aspects are interrelated. For example, low or limited concept definition is likely to be combined with low or medium generality, as is the case in RAPPID [158]. However, this is not always the case, For example, Tropos [23] is bounded to only BDI agents and it is still applicable in many application domains.

Evaluation Framework Aspects		Description/Ranking Values
Concepts	Concept definition	How is the approach characterised regarding restrictions in the definitions of agents and agent-based systems? [limited (\subseteq), bounded (\subset), open (\supset)]
	Design in scope	Does the engineering approach provide explicit support for the design phase of the agent-based system engineering lifecycle? [yes (\checkmark), no($-$)]
	Heuristics support	Does the approach provide formal techniques to support application of design heuristics? [yes (\checkmark), no($-$)]
Models	Organisational settings	Are organisational settings first-class design constructs? [yes (\checkmark), no($-$)]
	Collective behaviour	Are collective behaviours first-class design constructs? [yes (\checkmark), no($-$)]
	Non-functional aspects	Are non-functional aspects explicitly modelled and considered in the design of the agent-based system? [yes (\checkmark), no($-$)]
Process	Design perspective	What is the development perspective of the approach? [bottom-up (\Uparrow), top-down (\Downarrow), both (\Updownarrow)]
	Support for reuse	Does the approach provide guidelines and techniques to reuse existing design knowledge? [yes (\checkmark), no($-$)]
	Design automation	Do suitable formal underpinnings exist that can automate the design process to a certain extent? [yes (\checkmark), no($-$)]
Pragmatics	Generality	What is the generality (possibility of being applied to many application domains) of the approach? [low (\circ), medium (\oslash), high (\otimes)]
	Abstractability	Does the approach provide formal support for reasoning at different levels of abstraction? [yes (\checkmark), no($-$)]
	Tool support	Is there any assistance to the agent-based system designers by some software tool? [yes (\checkmark), no($-$)]

Table 3.1: Description and ranking of evaluation framework aspects

		RAPPID	DESIRE	Gaia	MESSAGE	Tropos	Zeus	KARMA
Concepts	Concept definition	\cong	\diamond	\times	\times	\diamond	\diamond	\times
	Design in scope	–	√	√	–	√	√	√
	Heuristics support	–	–	–	–	–	–	√
Models	Organisational settings	–	–	–	–	–	√	√
	Collective behaviour	–	–	–	–	–	√	√
	Non-functional aspects	–	–	–	–	√	–	–
Process	Design perspective	↓	↓	↓	↑↓	↓	↑	↓
	Support for reuse	–	√	–	–	–	√	–
	Design automation	–	–	–	–	–	–	√
Pragmatics	Generality	○	∅	∅	⊗	⊗	∅	⊗
	Abstractability	–	√	–	–	–	–	√
	Tool support	–	√	–	√	–	√	√

Legend

○ - low	\cong - limited	↑ - bottom-up	√ - yes
∅ - medium	\diamond - bounded	↓ - top-down	– - no
⊗ - high	\times - open	↑↓ - both	

Table 3.2: Comparison of ABS engineering approaches

3.2 Comparative Evaluation of ABS Engineering Approaches

A representative ABS engineering approach from each class of the classification scheme proposed in Section 2.4 has been evaluated according to the four views of the conceptual framework described in Section 3.1 (The approaches are reviewed in detail in Appendix A). A summary of the results is presented in Table 3.1.

Regarding the *Concepts* perspective, about half of the ABS engineering approaches (DESIRE, Tropos and Zeus) are bounded to specific agent architecture. RAPPID is the only one limited to specific agent types as well. Furthermore, the majority of the approaches examined (DESIRE, Gaia, Tropos, Zeus and KARMA) consider design as an explicit step in the ABS engineering

lifecycle. However, only KARMA/TEAMCORE provides formal support for heuristics in the design of the ABS. Clearly, this is a general deficiency of current ABS engineering approaches.

As far as it concerns the *Models* perspective, only Zeus and KARMA/TEAMCORE explicitly model organisational settings. Representing collective behaviours as first class design constructs is also not supported in most of the examined approaches. The only exceptions are Zeus where collective behaviours can be represented by role models and KARMA/TEAMCORE where collective behaviours are modelled by appropriate team plans. The lack of support for non-functional aspects is even more pronounced. Indeed, only Tropos considers non-functional aspects in the design of ABSs.

In the *Process* perspective, only MESSAGE/UML allows working in both top-down and bottom-up fashion and the current version of MESSAGE/UML supports only the analysis phase of the ABS engineering lifecycle. Zeus supports bottom up design, the rest of the approaches are all allowing top-down design. Furthermore, only two approaches explicitly provide support for reuse, DESIRE and Zeus. DESIRE includes guidelines about how the agent system designer can reuse generic task components in the design of the ABS and Zeus includes guidelines about how to reuse generic behaviours represented by role models and generic agent characteristics — for example negotiation strategies. There is also significant lack of support for automatic design of ABSs. Only KARMA/TEAMCORE supports automatic selection of the agents that will participate in the agent organisation based on team plans specified by the designer.

Regarding the *Pragmatics* perspective, approximately half of the approaches (MESSAGE, Tropos and KARMA/TEAMCORE) are general targeting a broad range of application domains. The rest are restricted as follows: Gaia assumes closed ABSs consisting of small numbers of static, cooperating agents. Zeus has restrictions regarding the environments where the agents produced can operate. For example, Zeus agents cannot be mobile and they require a large amount of physical RAM memory to execute. DESIRE is also specific to applications requiring static agents whose behaviour can be described by a task-based hierarchy. RAPPID is the most specific approach since it targets a specific application domain; that of supporting industrial product design.

The above analysis highlights certain weaknesses in existing approaches and it shows that there is no approach supporting all framework aspects. As a result, a number of issues that would require further research can be identified. These are discussed in more detail in the next section.

3.3 Implications for Further Research

The comprehensive analysis of existing ABS engineering approaches (see Sections 2.4.5 and 3.2) has demonstrated that none of the approaches reviewed covers all aspects of design support

included in the evaluation framework introduced in Section 3.1. An effective approach to ABS design should therefore cover a number of outstanding issues, which are described in more detail in the following sections.

3.3.1 Support for Design Heuristics

Design heuristics are considered very important for the engineering of robust commercial software [179]. Considering that the design process needs to be automated to a certain extent to reduce development effort, as discussed in [124], an effective design method should support the application of heuristics in both manual and automatic design steps. Automated application of heuristics requires that models with appropriate formal underpinnings need to be used in the design process.

Existing ABS engineering approaches do not provide systematic and rigorous models for considering heuristics in the design of the ABS. In approaches based on formal methods, such as DESIRE [22], software design heuristics can be taken into account in a rigorous manner but there are no guidelines and systematic methods to assist the designer in the application of heuristics. The designer needs to manually incorporate the heuristic rules in the formal ABS specifications.

There are some approaches, that provide some informal ABS design heuristics. An example is Zeus [147], where two design heuristics are provided: the *sphere of responsibility* and *point of interaction* heuristics. According to the sphere of responsibility heuristic, the designer should partition the application resources to areas of control and represent each area of control with a software agent. The point of interaction heuristic refers to representing each resource in the application domain with an agent. However, those informal heuristics cannot be easily applied to the design of large ABSs. Furthermore, it is difficult for the designer to predict the effect on design decisions when those heuristics contradict with other requirements (e.g. non-functional requirements). Indeed, often it is important for a method to formally combine design heuristics with application requirements [44, 225]. In this way, consistency checking would be done automatically by a software tool and design heuristics would be taken into account, to the extent that they do not conflict with other application requirements. This thesis, contributes to this issue by introducing a rigorous method to apply design heuristics. Its contribution is described in detail in Chapter 5.

3.3.2 Organisational Settings

As established in Section 1.3.2, the term organisational settings is used to refer to the general rules and conventions, as well as various authority relationships and coordinating interactions, that exists among entities in an organisation [73]. Organisational settings are important both to fully utilise the potential of an ABS [64] and to address challenging issues, including system

openness [220] and the dynamism of the environment [8]. Organisational settings are of particular importance when the ABSs aim to support the operation of human activity systems. In such cases, the organisational settings of the human systems should be aligned with the organisational settings of the ABSs [96, 98].

Some agent system engineering approaches explicitly model organisational settings of the ABS — for example, MAS-CommonKADS [91] and SODA [150] — and there are cases where the agent organisation is designed during a distinct design step, before the agent behaviour is completely specified [7]. However, it is argued that even when organisational settings are explicitly modelled, the models only represent the organisational relationships between agents without considering social tasks and social laws [222]. Furthermore, organisational settings are not considered as first class design constructs apart from a few exceptions of approaches based on role modelling [147, 150]. Another problem concerning organisational settings is that existing approaches do not provide any rigorous methods for combining organisational settings with application functionality. This has to be done intuitively by the designer without any assistance by a software tool.

Considering the above discussion, it is apparent that a rigorous method to represent organisational settings and combine them with application functionality, while considering them as first class design constructs, is required. This representation should not only model the organisational relationships among agents, but it should also allow modelling of social tasks and social laws. This would significantly contribute towards addressing the openness and the dynamism of the environments where real-world ABSs must operate. This thesis contributes towards this objective by introducing an approach based on role modelling. This approach is described in detail in Chapter 5.

3.3.3 Collective Behaviour

A similar problem exists regarding representing collective behaviour. In this thesis, the term *collective behaviour* is used to refer to behaviour which results from the interaction of a number of entities in a particular context. Many authors argue that collective behaviours should be treated as first-class design constructs, namely that they should be able to be instantiated and given identity [2, 111]. However, even where this issue is addressed, such as in Zeus [38], there is no rigorous way to reuse collective application functionality and combine it with organisational settings.

Clearly, representing collective behaviours in a rigorous manner so that they can be directly combined with organisational settings is a research issue of major interest. A prominent direction toward this goal is to model collective behaviour using role models [49, 51, 111] and this is the direction followed here. Furthermore, this thesis introduces the necessary formal

underpinnings so that collective behaviours and organisational settings can be combined in a rigorous manner.

3.3.4 Non-Functional Aspects

An issue of major concern in ABS design is the modelling and consideration of non-functional aspects such as security and performance [18]. To achieve this it is necessary to explicitly model and consider non-functional aspects before actually deploying an ABS. Treatments of non-functional aspects can be classified as *product-oriented* and *process-oriented* [35]. Process-oriented approaches develop techniques for justifying decisions concerning support of non-functional aspects during the software development process. For example, adding elements in support to particular non-functional aspects at each modelling step when creating design models. Product-oriented approaches deal with non-functional issues from the evaluation point of view. They involve examining software products to check if they fall within certain constraints of non-functionality and amending them as needed. Ideally, elements from both approaches should be combined to support explicit modelling of non-functional aspects since they complement each other [145].

To the best of author's knowledge, no other ABS engineering approach explicitly considers non-functional aspects in design apart from Tropos [23], which, at some stage, includes introducing actors and sub-actors that contribute positively to the satisfaction of non-functional requirements. However, the Tropos approach to modelling non-functional aspects suffers from two main weaknesses. Firstly, it models non-functional aspects in a way that it cannot be directly reused in other ABS designs. Secondly, quantitative characterisation of non-functional aspects is not possible.

In some cases, non-functional aspects are the basis for criteria for reorganisation in dynamic approaches, as is the case in KARMA/TEAMCORE. In these instances non-functional aspects are taken into account by adjusting the agent behaviour and the organisation of the ABS on run-time. However, this treatment of non-functional aspects impedes the reuse of non-functional models. It also contributes to significant consumption of resources and system instability.

A new approach to ABS design should provide explicit models of non-functional aspects that would be used on design time. Furthermore, modelling of non-functional aspects should combine both product-oriented and process-oriented approaches. In addition, models of non-functional aspects should be able to integrate with models of system functionality in a rigorous manner. In this thesis, these issues are addressed by modelling non-functional aspects using explicit role models and constraints on role characteristics.

3.3.5 Automating the Design Process

In order to reduce development effort and software design errors the design process should be partially automated [124]. This view is also adopted by informal ABS engineering approaches [50, 186, 205] that try to provide the formal underpinnings for automatically designing ABSs from appropriate informal specifications. The common way of doing that is by progressing from analysis to design by successive formal transformations of the analysis models. The transformations used, however, focus on ensuring that the designed agent components are correctly represented in respect to the analysis models, using object-oriented software engineering concepts and techniques. For example, in [186] formal transformations are used to decide on the number of objects and concurrent threads that should be used to correctly realise the behaviour of each agent component. To the best of author's knowledge, current informal ABS engineering approaches do not provide any automatic support for actually deciding on what behaviour each agent in the ABS should have. This is not the case for dynamic approaches where the design of the agent system is done during reorganisation steps. For example, in KARMA/TEAMCORE [192] the agent components are automatically selected based on specifications of the agent-based application requirements described in the STEAM modelling framework [191] (see also Section A.7). However, KARMA/TEAMCORE assumes that agents already exist in the cyberspace, which is not generally the case. In addition, the focus of this work is on automating the static design process which is done only once before deployment.

To provide support for automatic design of the agent-behaviour based on informal models, the required functionality and the criteria used to determine the behaviour of each agent should be specified with appropriate rigour. Furthermore, a systematic design process with clear manual and automatic steps is required. In this thesis these issues are addressed by formalising role relations with respect to allocating roles to agents and by introducing a design process based on the principles of synthesis.

3.3.6 Working at Different Abstraction Levels

Design complexity needs to be reduced in order for an ABS engineering approach to be easy to understand and apply [169]. Alagar and Periyasamy in [1] stress that the most common and effective technique for dealing with complexity is *abstraction*. Abstraction means generality. For example, to describe a collection of similar objects having some common attributes one can use the notion of *set*. In this way it is possible to unambiguously refer to all objects in the set at the same time without concern about how the set is represented.

There is a consensus that abstraction in software design reduces design complexity [139]. It is mandatory to deliver abstraction mechanisms to programmers both in software engineering methodologies and in programming languages [183]. Although abstraction has the trade-off of

reducing software efficiency and performance [40], it may add to the reliability of the produced software as frequently used components are thoroughly tested and the design process can be automated [1].

Abstractions in software specification can be achieved in two main ways [1, 183]:

- By partitioning the world of objects using modular decomposition techniques. This allows us to understand the individual and the collective behaviour of objects at various levels of detail. For example, appropriate role models can represent groups of objects interacting for the same purpose [108]. Then one can reason at the individual role level or at the role model level depending on the detail required.
- By encapsulation of related functional characteristics to well understood models. For example behaviours related with a position and a set of responsibilities can be represented by appropriate roles [111, 209, 224]. It is argued that reuse of appropriate software components is mandatory to efficiently engineer ABSs for real-world applications [81, 193].

As abstraction is a common practice in software design, a number of ABS engineering approaches allow the designer to work at different levels of abstraction. However, not all of them provide appropriate formal support. For example, MESSAGE/UML allows modelling at levels 0 and level 1 but there is no formal description of the relations between the models of the two levels. As a result, proper use of MESSAGE/UML requires the designers to have a clear understanding and explicitly consider the links between models at levels 0 and 1, which makes the ABS design task more difficult,

The only approaches examined in Section 3.2 that provide formal support for working at different levels of abstraction are DESIRE [22] and KARMA/TEAMCORE [192]. However, their support is limited. DESIRE only supports interaction between tasks at different abstraction levels and KARMA/TEAMCORE supports teamwork at different levels of abstraction in the form of joint intentions. Agent behaviour, however, is characterised with other aspects as well. For example, coordination protocols or negotiation strategies, which the designer should specify at the lowest level of detail in those two approaches. This problem is addressed in the Zeus approach [147]. For example, in Zeus the agent system designer can either select a predefined negotiation strategy or specify all negotiation rules in detail. Zeus models agent behaviour at different levels of abstraction based on role modelling. However, this support is informal since the relations among roles have not been given formal semantics.

Consequently, none of the approaches examined provides adequate support to the designers for working at different levels of abstraction and this is therefore an open issue. Different levels of abstractions should include all aspects characterising agent behaviour, such as goal-based behaviours and coordination protocols. This thesis contributes towards this issue by extending

the Zeus role modelling approach to include formal semantics of relations among roles and more characteristics in the role definition.

3.4 Summary

This chapter proposed a framework to assess ABS engineering approaches with respect to design. Using this framework, a number of approaches have been examined revealing a number of issues that would require further research.

The proposed framework suggests looking into ABS engineering approaches from four views: *Concepts*, *Models*, *Process* and *Pragmatics*. The *Concepts* view refers to the modelling concepts used to model ABSs and it concerns the generality of the concept definition, the existence of specific support for design in the ABS engineering process and the support for design heuristics. The *Models* view refers to modelling of organisational settings and collective behaviour to be used as first class design constructs and to explicit modelling of non-functional aspects. The *Process* view examines the perspective of the design process and whether it can be based on reuse and if it can be automated. The *Pragmatics* view evaluates the applicability of the approach to real-world applications by assessing the generality, the complexity handling and the tool support of the approach.

The evaluation of a representative ABS design approach for each class in the classification scheme introduced in Chapter 2 reveals considerable weaknesses in current approaches with respect to designing the ABS. Existing approaches do not provide formal support for design heuristics, do not consider organisational settings and collective behaviour as first class design constructs and they do not take non-functional aspects into account in the design. Overall, current approaches cannot automate the design process to any extent and they do not provide adequate support for working at high levels of abstraction.

An effective semi-automatic approach to ABS design should address these problems, and thus satisfy all the criteria of the evaluation framework proposed in this chapter. In particular, such an approach would extend informal ABS design approaches based on role modelling by providing the formal underpinnings for design heuristics support, design process automation and work at high levels of abstraction. Furthermore, the proposed approach would include modelling mechanisms that would enable considering ABS organisational settings and collective behaviour as first class design constructs and would allow taking non-functional aspects into account in ABSs design. Finally, this approach should be implemented in a software tool.

Chapter 4

Using Role Modelling for ABS Design

This chapter reflects the need for a systematic approach to ABS design based on role modelling. It is argued that such an approach should combine role-modelling aspects from sociology and information systems engineering and that it should be grounded on a comprehensive social system analysis theory such as role theory. Finally, directions on how role modelling can be used to address the open research issues raised in Chapter 3 are highlighted paving the way for a detailed discussion of the proposed ABS design approach in Chapter 5.

4.1 Complete Role Modelling Approaches

Role concepts are used in many areas of computing, such as object oriented software engineering [163] and workflow system modelling [219]. However, in many cases the use of role concepts is done in an ad-hoc manner. For example, the term ‘role’ is not defined in detail, as is the case in [64], or no criteria for deciding what roles each entity in the system can play are specified [2]. Ad-hoc use of roles makes it difficult to design and implement software systems systematically and such approaches are considered *incomplete* [126, 127]. In contrast, when role modelling approaches are clear from ambiguities and include appropriate methods that can assist in the analysis, design and implementation of role-based software, then they are termed *complete* [127].

Complete approaches should provide a clear description of the term ‘role’ and a distinction between various role types if applicable. Furthermore, they should describe how roles are identified, what relationships can be established between various roles or types of roles and what inconsistencies may arise in role specifications. Finally, they should define how roles can be assigned to software components.

These criteria are used to establish the foundations of a role modelling approach for ABSs design by examining aspects from uses of role modelling in social and business system analysis and software systems engineering. This prepares the ground for a more detailed discussion about the ABS design method proposed in this thesis, which is positioned in the next chapter.

The discourse starts by defining the term ‘role’ and its characteristics. Subsequently, an overview of *role theory*, a comprehensive theory that models and studies social systems based on role concepts, is given. In Section 4.3, various approaches to using role modelling in software systems engineering are classified and discussed. Section 4.4 contains a discussion of

how role modelling can address the issues for further work in supporting ABS design raised in previous chapter, Finally, Section 4.5 summarises the issues discussed throughout the chapter.

4.2 Modelling Social Behaviour Using Roles

As mentioned in Section 2.2.5, roles are representations of behaviour. In this thesis, the interest is in behaviour in the context of social systems since the view is that ABSs should be aligned with the human activity systems they support. However, in social systems roles can be used to refer to different facets of social behaviour at different levels of detail. Therefore, defining roles to represent social behaviour is not trivial. This section discusses how roles and role characteristics are defined in the area of sociology. The emphasis of the discussion is placed on role concepts that would be useful in defining roles, and using role modelling for ABS design considering the completeness criteria proposed in Section 4.1.

4.2.1 Defining the Term ‘Role’: a Social View

The term ‘role’ has been extensively used to describe social behaviour in the areas of sociology and social psychology. A role describes behaviour within some social activity context and in connection with relationships with other roles. In order to be useful in modelling and designing ABSs a role definition should lie within a context and be focused on the normative aspects of social behaviour.

4.2.1.1 Social Aspects of Role Definitions

Although the origins of the term “*role*” can be traced back several centuries, it first appeared with a meaning close to the one it has today in the theatre where actors “play roles”. Roles have been broadly used in the area of sociology to model individual behaviour as well as social system structure and organisation [6, 16], and to provide the theoretic basis for discussing about system creation and system evolution [6, 15]. There are numerous definitions of the role concept in sociology, all referring to behaviours of persons in some context. Biddle [15] summarises some representative definitions of the role concept in sociology as:

- “*what the actor does in his relations with others*” ([153], p.25);
- “*what persons do as occupants of the position*” ([142] p.280); and
- “*what the actor does ... seen in the context of its functional significance*” ([153], p.25).

Biddle also provides a similar definition of a role as “*a characteristic behaviour of one or more persons in a context*” ([15] p.58). A comprehensive historical review of the evolution of the definitions of the role concept in sociology can be found in [196].

Several authors have proposed to start from sociological definitions of the term role and define roles to represent agent behaviour [122, 201]. Similarly, many authors in the agent systems research, e.g. [83], view roles as primary sociological concepts that must be redefined in an operational manner to be useful for modelling behaviour in ABSs. A definition of the term ‘role’ specifically for modelling agent behaviour is presented in [201], where a role is defined as “...*the functional or social part which an agent, embedded in a multi-agent environment, plays in a (joint) process like problem solving, planning or learning*” thus focusing on operational aspects. Lind ([122], p. 17) presents a different focus for as a social construct by stating that: “*A role is a collection of expectations towards the behaviour of the inhabitant of a particular position that allows the members of the society to predict the inhabitant's behaviour and to plan according to their expectations*”. Werner [202], on the other hand, limits the concept of role to purely cognitive states that are determined by the knowledge, the permissions, the responsibilities and the assessment of the current situational context of the agents. Finally, Gasser [73] defines the concept of role as a “*prototypical type of behaviour*”.

Generally, it is agreed that a role corresponds to a position in a social structure. A person or an agent can be associated with more than one social position. The actual behaviour of a role can be related to the individual's own ideas of what is appropriate (*role cognition*), to other people's ideas about what he will do (*expectations*) or to other people's ideas about what he should do (*norms*) [6], p. 29. In this light, a role may be understood as *a set of norms and expectations assigned to a social position in a particular context*, an approach that will be adopted in this thesis as well.

4.2.1.2 Role Relationship Zones

The context in which roles are defined concerns both the physical environment as well as the social environment including other roles as well. Therefore, the behaviour represented by a role can be related to that of other roles to form a “*social net of role relationships*” [59]. For example, two roles are related to each other when they interact in some way. Role relationships have many subjective aspects, for example, they may reflect the need of individual workers for informal communication in working environments [117]. Therefore, it is necessary to identify the normative parts of role relationships that can be modelled or formalised. Elliot [59] distinguishes several “*zones*” or “*levels*” at which role relationships can be modelled (Figure 4.1). The work described in this thesis relates essentially to the legal and institutional zone where the obligations, rights and protocols for required inter-communication among roles are defined. For example, when a participant in an auction submits a bid to the auction coordinator this must be done according to a predefined and specific protocol, which leaves no ambiguities regarding the amount bided and the method of payment. Interactions in the inter-subjective zone are essentially informal communications [59]. As they do not have the mandatory aspect of a

protocol it is not practical to attempt to support such interactions in an ABS [127]. Furthermore, the current trends in agent research indicate that agents will always communicate according to specific communication protocols specified by standardisation bodies, e.g. FIPA [67]. Therefore, modelling role behaviour at the inter-subjective level is outside the scope of this work.

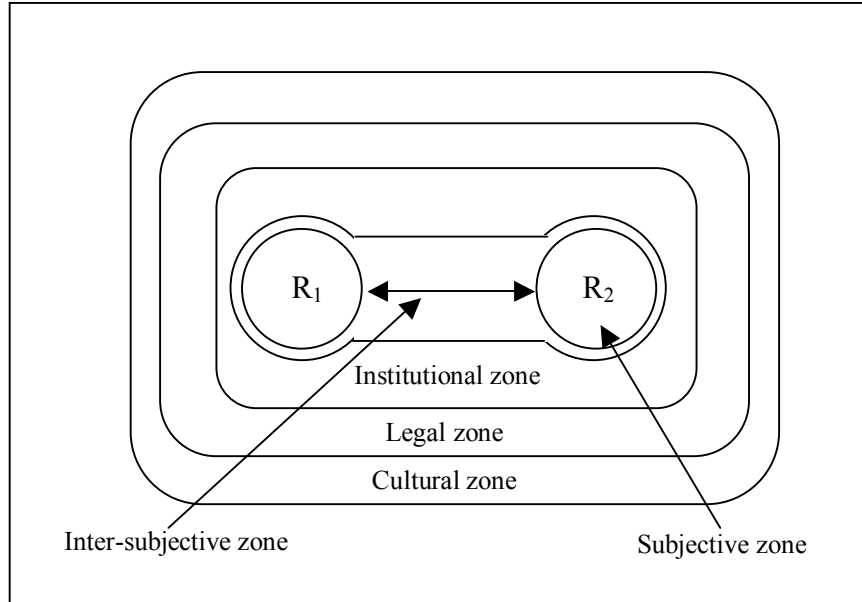


Figure 4.1: Role relationship zones

4.2.2 Overview of Role Theory

Role concepts have often been used in sociology in an ad-hoc manner. To be practically applicable in the design of ABSs and fulfil the criteria proposed in Section 4.1 role modelling should be based on a systematic theory. *Role theory* is a comprehensive theory for describing and reasoning about social behaviour using roles. It provides the systematic approach and formal definitions necessary for the application of role modelling to agent system design. However, role theory still needs improvement as far as it concerns formalisation of role relations and assignment of roles to agents.

Role theory [15, 16] is a science concerned with the study of behaviours that are characteristic of persons within contexts and with various processes that presumably produce, explain or are affected by those behaviours. Those behaviours are represented with appropriate roles. In this view, individuals in a society are expected to fulfil certain *roles* (e.g. father, director, doctor) that predefine their rights and duties in that society, in the same way that actors play a part. The behaviour of roles is characterised by *authorities* (rights) describing things that can be done and *responsibilities* (duties) describing things that must be done. For example, directors, help-desk staff, developers and test engineers are all associated with job descriptions specifying their

responsibilities in a business organisation. Role theory has the advantage of being a complete science with its own terms and concepts.

Role theoretic terms are intuitively understandable since they are also used in common language. However, role theoretic concepts lack adequate formalisation, something that is considered a significant disadvantage [16, 196]. This is particularly problematic if role theory is to be used for the modelling of artificial societies. In this thesis, this problem is mitigated by introducing a formal model of role relations and a method for automatic allocation of roles to agents.

4.2.3 Role Theoretic Concepts

To facilitate subsequent discussions, the basic concepts of role theory are presented in more detail together with implications about how these concepts can be used for applying role-theory to ABS design:

- **Person:** For the sociologist, a person is an entity that exhibits behaviour normally as a member of a community. A person is a carrier of culture and a representative of the assumptions and values of the community he/she is a member of. In addition, a person receives stimulus from the social environment and responds to challenges with behaviours that are functionally linked to other behaviours for the accomplishment of *tasks*. Furthermore, the behaviour of a person is also dependent on the *goals* the person has, for example one that seeks to increase his personal income is likely to work overtime. Agents also try to achieve goals, can carry out tasks and exhibit their behaviour in the context of ABSs which can be viewed as organised social systems [64, 73, 98, 150, 221]. Therefore, role theory can be applied to ABS modelling and the person concept can map onto the concept of autonomous agent. The goals and tasks of a person defined in role theory correspond to agent goals and tasks as they have been commonly defined in the agent literature (see also Section 2.2.2).
- **Roles:** Roles are patterned human behaviours that are the basis for describing the behaviours of persons in a society. Personal roles that are commonly played by sets of persons are termed *identities*. For example, the roles describing the behaviours of the relatives of a person are identities. Therefore, the behaviour of a person is more or less known once his/her identity is known. Furthermore, roles can be associated with *social positions* (or *statuses*). In general, a social position is an identity (a characteristic role) that designates a commonly recognised set of persons. For example, the terms ‘physician’ and ‘university lecturer’ both constitute a social position.

Agent roles should be defined in a similar manner, extended as appropriate to meet all modelling requirements, i.e. non-functional aspects.

- **Role functions:** Roles are likely to have characteristic effects, or functions, within the social system. The effect of role functions has a purpose aligned with the overall goal of the role. For example, the physician who wears a white coat in the hospital helps others to recognise him or her quickly and thus follow his or her orders in an emergency. Agent roles can also have functions from a role theoretic perspective. Those functions can be modelled as appropriate role characteristics, for example the coat colour could be a characteristic of the physician role. This approach to modelling role functions is discussed in more detail in Section 5.2.
- **Context:** There is a consensus that most role behaviours are contextually bound. For example, a football match and a church service are contexts and an audience member is a role describing the behaviour of the audience. Clearly, the audience behaviour, for example whether to cheer or to sit in solemn silence, depends on the context, stadium or church, the person has entered. Hence, context affects role definition. The agent roles are defined in the context of the application domain the ABS is targeting.
- **Social Systems:** According to role theory, roles are normally imbedded within social systems and role concepts may easily be used for the analysis of complex social forms. For example, most factories have a table of organisation that lists the social positions of its employees. Each of these positions is assigned a job to do, and each exhibits characteristic role behaviours. The roles of the various positions are *specialised* and *interdependent*. In a production line, for example, performance of several thousand roles may be necessary to generate an automobile, a vaccine or a computer. In such a context, individuals must learn to accommodate a specialised role if they are to remain members of the organisation. Social systems correspond directly to ABSs since ABSs are by definition agent societies.
- **Role assignment – Socialisation:** According to role theory, roles are assigned to persons through the sharing of *expectations* of a particular behaviour in a process termed *socialisation*. In other words, those that exhibit the role are stimulated to do so because they learn what behaviours are expected from them. For example, it is said that physicians wear white coats in the hospital ward because they have learned that their patients have such expectations. However, in ABS design role assignment is a characteristic of the approach followed, e.g. static or dynamic. In the method proposed in this thesis, role assignment is done statically at design time. Role assignment is further discussed in Section 5.2.7.
- **Role differentiation and specialisation:** Two or more roles are differentiated if they have but few behavioural elements in common. *Role differentiation* can be used not only to separate performances of persons who occupy different social positions, but also behaviours

of a single person in various contexts. When different persons perform differentiated roles of the system, this is termed *role specialisation*.

- **Role Integration:** Several terms have also been suggested for describing the ways in which roles in a social system relate together. The general term describing a well ordered social system is role integration. When a role system is role integrated, this means that its roles fit well together. There are various ways in which *malintegration* may be generated. Performers of different roles may find their duties overlap, that their roles are functionally interrelated although they have inadequate means of communicating, that they are competing against one another for scarce resources needed for role performance, or that differing standards of reward or demand apply to their various roles and so on. Role integration is associated with role dependency relations, which are discussed further in the next Section.

4.2.4 Role Dependency Relations

There is a consensus in the sociology literature that mutual dependency relations can be formed between roles [6, 16]. Those dependencies affect the existence and compatibility of roles both in the same social system and in the same person. That is some roles can only exist if other roles exist as well. The role of a “*physics teacher*” only makes sense if the corresponding role of (at least one) “*physics student*” exists in the same social system as well. An example of role incompatibility is a university examination. The same person cannot be both taking the exam and invigilating at the same time.

Biddle [16] emphasises that dependency relations between roles can describe particular aspects of social behaviour. For example, entry to some social positions is governed not by abilities or desires of the person, but rather by accidents of birth or customs of the society. An example of this is when women and or persons from ethnic minority groups are denied opportunities for employment although they are fully qualified. Moreover, positions are sometimes arranged in an age or achievement-graded sequence such that the person must first achieve membership in a given position of the sequence before he or she can be considered for elevation to the next position. For example, those without a bachelor’s degree will not normally be accepted for postgraduate education in a university.

Dependency relations among roles can be also used to describe problems that may be encountered in particular behaviours of individual persons [6, 15]. Some roles are difficult to perform and take great natural ability or years of practice to learn. Some times the person is subjected to incompatible role expectations (or *role conflict*) wherein he or she is required to do two or more things that cannot all be done. In addition, individuals may suffer from *role overload* when too much is asked of the person, as may be the case for persons who have to

both work and study at the same time. Furthermore, sometimes the role the person is asked to perform is *inconsistent* with his or her needs or basic values. For example, it is not morally acceptable for militants to release national secrets to other nationals. In addition, the behaviour of a person can be considered *deviant* by the society and the person is subject to punishment or institutionalisation until the problematic roles are replaced. For example, theft is punished in most modern societies.

Dependency relations among roles are therefore an important instrument for describing behaviour at both individual and organisational levels. The power of this instrument can be increased if it is combined with appropriate role selection in the application context of interest, and with appropriate specification of role characteristics to reflect the application requirements. For such an approach to work, however, role relations need to be defined within a formal system to enable reasoning regarding expected agent behaviour. This definition is one of the main contributions of this thesis and is discussed in Section 5.2.7.

4.2.5 Role Identification and Role Types

A major problem in the field of sociology is the delimitation of roles that occur within a society. Role theory addresses this issue by considering two broad criteria for role identification [16]: (1) Roles may be associated with persons or within contexts; and (2) a role may also be defined in terms of its content or a function that is performed by the role. Furthermore, role theory considers the following role types that can be used as criteria for social role identification:

- *Species roles*: Some roles are characteristic of human beings as species. For example, human beings characteristically walk on two legs, breathe, and ingest food through their mouths. In the same way, agents communicate in a standard communication language and are executed in a particular host each time.
- *Person-associated roles*: One of the simplest ways of defining roles is in terms of behaviours associated with a specific set of persons. These roles can be: (1) *Societal roles*, which are patterns of behaviour that are characteristic of persons who are members of a given society, such as the English speaking citizens of some country. (2) *Positional roles* representing behaviours characteristic of those sharing a commonly recognised identity or social position, for example, policemen. Finally, (3) *Personal roles*, which are the behaviours characteristic of an individual, for example, the role of a known politician in the political affairs of a country.
- *Contextual roles*: Roles can be defined in terms of context and various contextual cues may be associated with certain roles. In particular, roles can be associated with two main types of contexts: (1) The *physical context* or *setting* in which the behaviour takes place. For

example people normally drop their voices when entering a dark room; and (2) the *activity* in which roles can occur, for example a football game, or an orchestral concert.

- *Functional roles:* As noted in Section 4.2.3, roles can accomplish functions. When a role contributes to two or more distinct functions, it can be partitioned into its functional components. For example, it is possible to establish those behaviours of the teacher's role that contribute to "*instruction of pupils*" versus those connected with "*pupil counselling*". Those functional components can be considered as separate roles themselves. Functional roles are used to limit the concept of a role to represent only a limited range of behaviours. Functional roles can be further specialised in specific domains. For example, they can be occupational, recreational or economic roles. Such uses illustrate roles that are content-specific.
- *Task-based roles:* Some social systems include tasks. For example, in a modern organisation tasks are often found to be assigned explicitly to each position making up the complement of positions in the organisation. Task based functional description is a convenient approach for modelling the normative behaviour of agents in ABSs.

Generally, roles can be identified through several criteria simultaneously. A general approach to role identification in social systems is to determine the particular role types existing in a social system and further specify the roles involved considering the context and the functions associated with the role types [16]. In developing agent applications the system designer is faced with a similar problem in identifying coherent sets of behaviours that can be grouped together to form the roles that occur in the problem domain. In this thesis the general role identification approach suggested in role theory is refined to make it suitable for ABS design.

4.3 Using Roles in Information Systems Modelling

The concept of role has been extensively used in information systems engineering as a primary construct for building conceptual models. In this section, the ways that roles and role modelling are used in the areas of business process modelling, distributed systems management and object oriented software engineering are discussed. This discussion reveals interesting issues that should be taken into account when applying role modelling to the design of ABSs, particularly when they are viewed in conjunction with the role theoretic concepts discussed in Section 4.2.3.

4.3.1 Roles in Business Systems Modelling

Roles in business systems have been used to model the behaviour of actors participating in business processes [115]. The applicability of roles has been demonstrated in many areas where business process modelling is needed including workflow management [219] and business process re-engineering [151]. Role concepts have been defined in a manner similar to that of

social roles described in Section 4.2; however they differ in many aspects the most important being that they focus on modelling only normative parts of business behaviour, namely behaviour that is exercised in a predefined manner, for instance an auction negotiation.

In the majority of business system modelling approaches, roles are used as a link between other business modelling concepts. For example, in a typical workflow modelling approach, such as the one described in [133], roles link tasks and agents. Workflow processes consist of tasks and each task represents a business activity, for example, producing an electronic insurance quote. Tasks are associated to roles and roles act as placeholders for agents. Agents that fulfil a role perform the tasks that are associated with it. The main advantage of using roles as placeholders for agents is that assignment of agents to roles and of tasks to roles can be done separately. Therefore, agents can be dynamically assigned to or removed from roles at run-time.

A workflow approach focusing on dependencies between roles is the *Actor-Dependency* model proposed by Yu in [217, 218]. The approach uses the term *actors* to refer to roles, positions and agents. Agents *play* roles and *occupy* positions that *cover* several roles. Further modelling concepts include goals tasks and resources. Actors have goals, execute tasks and have access to resources. Actor dependencies can be based on task dependencies, resource dependencies, or goal dependencies. An overview of the Actor-Dependency model is given in Figure 4.2.

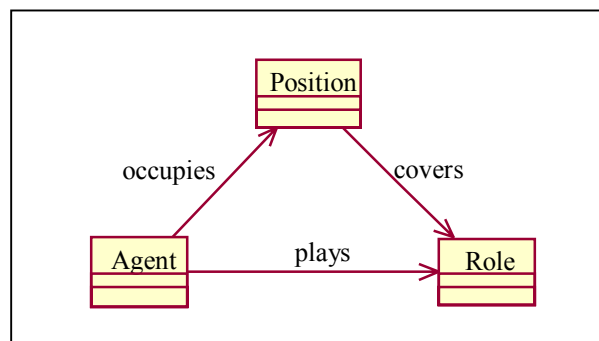


Figure 4.2: Agent-Position-Role dependencies in the Actor-Dependency model

Bubler [27] suggests that roles should be defined as a set of *capabilities* the role offers for the accomplishment of tasks and a set of *requirements*, which active business objects must fulfil in order to play a role. A “role resolution” process selects the set of active objects having the necessary capabilities to fulfil the roles. This process also takes into account predicates specifying additional constraints.

In a similar manner, Ould [151] defines a role as “a set of activities which, taken together, carry out a particular responsibility or set of responsibilities” and presents roles as basic concepts in business process modelling. That approach further introduces the existence of role *types* and *role instances* in business process models, and that these roles are filled or played by actors. In

addition, business process roles can be transferred, shared, or distributed between actors. Roles represent behaviour associated with functional groups, for example a support engineering group with functional positions — such as GUI programmer — with rank or job titles — manager — with class or types of persons — field worker, and with abstractions that correspond to descriptions of some work activity that can be performed by more than one job title, such as web page authoring.

It can be observed from the above discussion that a primary difference between role definitions in sociology and in business systems is that the definitions of roles in business systems focus on the normative aspects of business behaviour. This view is also adopted in this thesis as this work aims to support designing operational systems and, therefore, non-normative behaviour is not relevant. Instead, it is assumed that users of those systems will exhibit non-normative behaviour (for example, personal socialisation contact) without explicit support from the ABS.

In addition, some approaches, for instance [151], consider dependency relations between actors in modelling the business system. However, no method that can be used to specify how the agents should cover the positions and play the roles is given. Bubler [27] does offer some role resolution process but his role definitions are narrow and he does not consider dependency relations among roles. Furthermore, he assumes that appropriate objects always exist somewhere in order to fill in the roles, which is not always the case.

Modelling normative aspects using roles at different levels of abstraction and providing a systematic mechanism for allocation of roles to agents based on a rigorous model of role relationships is mandatory for reducing the complexity of ABS design. Therefore, those issues represent the focus of the work done in this thesis.

4.3.2 Role-Based Access Control in Distributed Systems Management

Many authors consider roles as an appropriate modelling construct to represent access privileges in distributed computer systems, for instance [126, 170]. Lupu et al., [126] emphasise the use of managerial roles for distributed systems access control. In that approach a role is defined within a domain (organization), and the domain has *policies* that determine *authorities* and *obligations* for its member roles. In particular, a role represents a *position* within an organization, and it has *responsibilities* made up of *activities* and required *interactions* with a number of related roles. Access control roles can be assigned to both human and software agents.

The components of a role according to [126] are shown in Figure 4.3. For each related role, a role has *obligation* and *authorization policies*, *concurrency constraints* and an *interaction protocol*. Authorizations stipulate which roles are under the authority of another role. Concurrency constraint specifications describe the parallelism and synchronisation between the

activities within and between roles. Conflicts can also occur, and one source is due to overlap in policies with respect to authorizations and obligations.

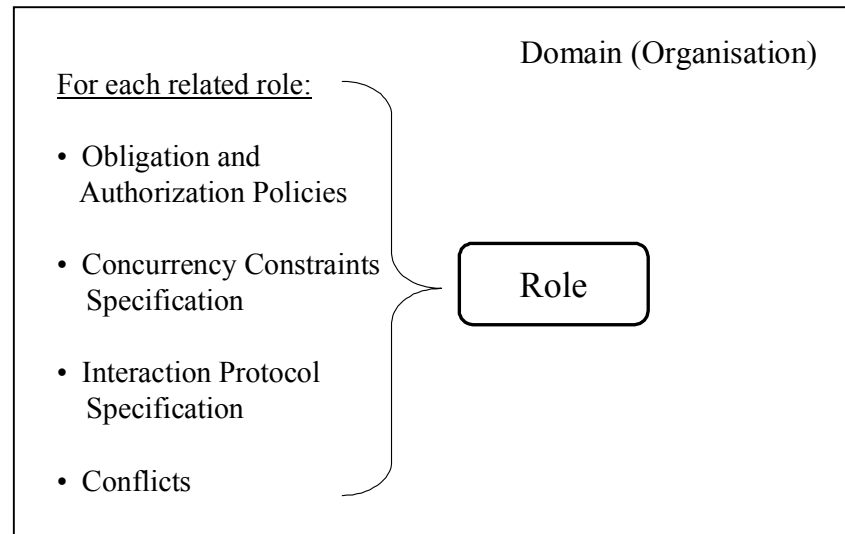


Figure 4.3: Role characteristics for distributed systems access control

Policies can be specified at various levels of abstraction, and, according to [126], policies can be detailed enough to specify the actions which represent the behaviour of a role. If a role has the proper authority it can delegate its duties or access rights to another role. Policies can also be propagated to sub-domains of a parent domain. Lupu [127] further identifies different relationships between roles, including hierarchical, resource sharing, information access, coordination and contractual. Those relationships are taken into account when specifying access control constraints.

Role modelling for access control progresses further in the direction of using role relationships for describing the system behaviour. However, such approaches are limited in the sense that they focus only on the access privileges of (human or automated) agents. For example, they do not model proactive, goal-oriented behaviour. In this thesis, role relationships are treated in a similar manner but the proposed role definition is extended to model sophisticated, both proactive and reactive, agent behaviour.

4.3.3 Roles in Object Oriented Software engineering

In role-based software engineering some of the weaknesses observed in role based business system modelling are addressed. The context is taken into account in role definition at low granularity and role relationships are systematically considered in the role allocation process.

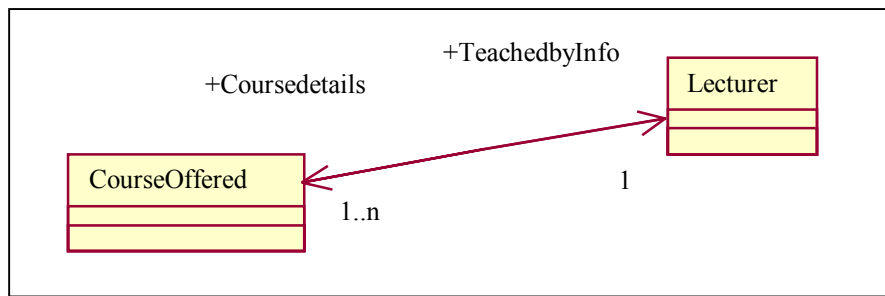


Figure 4.4: Roles as association names

4.3.3.1 Defining Roles in Object Oriented Software Engineering

Although roles have been associated with many different properties in different application contexts, the number of substantially different definitions is small. Role definitions can be classified according to four possible views:

Roles as *named places in class relationships*. In this case, a role is a name (label), which uniquely characterises a participant within an association. This type of role definition had been supported by many early object oriented analysis and design approaches, e.g. OMT [172], and it is also supported by UML [149]. For example, in Figure 4.4 *CourseDetails* and *TeachedbyInfo* are roles representing the interfaces of the classes *Courseoffered* and *Lecturer* in their association relationship. This type of role definition represents only static aspects of behaviour and describes only associations. Hence, roles defined in this manner cannot be reused in other classes. Therefore, a richer role definition is required to represent agent behaviour.

Roles as *patterns of interaction among objects*. In this view, roles are used to describe patterns of interaction among objects. A role is characterized by its attributes and the messages it may receive or send to, other roles. This view has been supported by many early object oriented methodologies, for example OOram [163]. It is also supported by UML [149], where roles can act as type specifiers in the scope of a collaboration diagram among objects.

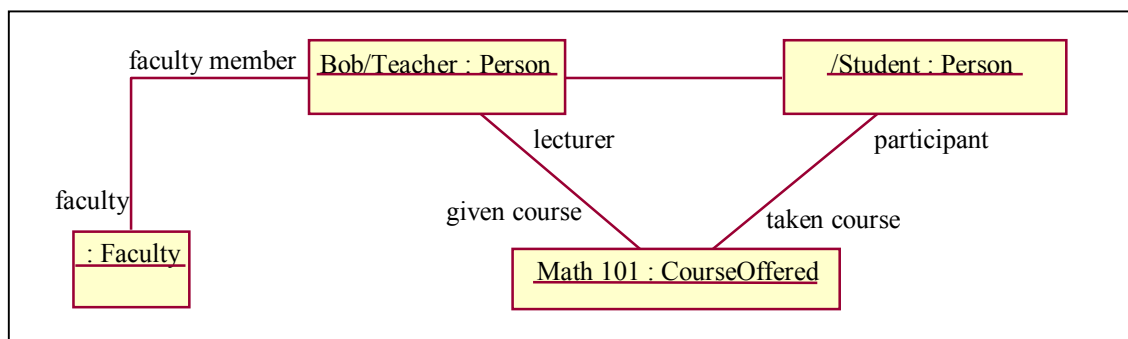


Figure 4.5: Roles as patterns of behaviour

In that case, roles are called *classifier roles*. For example, Figure 4.5 represents a collaboration diagram including six association roles (*faculty*, *faculty member*, *lecturer*, *given course*, *participant*, *taken course*) and two classifier roles (*Teacher*, *Student*). The notation used is the standard UML *objectname/role:class* syntax. For instance, *Bob* is an object of type *Person* that plays the role *Teacher*. The role *Teacher* represents all attributes and all messages sent and received by instances of class *Person* (i.e. *Bob*) that are relevant to the collaboration depicted in Figure 4.5. This type of role definition has a similar problem as the previous one. That is, roles are defined in terms of collaborations of particular classes and are therefore not re-usable elsewhere in the system. This limitation makes the UML role concept clearly not applicable for designing agent organisations and a more general definition of role is needed.

Roles as a form of *generalisation and/or specialisation*. This role definition is adopted when the lifetime of software objects is long and therefore their behaviour needs to be changed throughout it. In such cases, the view is that objects should obtain different roles throughout their lifetime as needed. Wong [204] describes an approach concerning object-oriented database engineering where roles represent generalisations or specialisations of objects. In that approach, roles can be played by objects or by other roles. Classes can be specified for objects or roles. An object class encapsulates the persistent properties of an object, while role classes define the transient properties. Role classes can be optionally restricted in terms of the type (class) of the object that can play role instances created from that class. The overall object model includes object classes and role classes linked by two types of relationship. The *is_a* or subclassing, which can be only between roles, and the *played_by* relationship, which can be both between roles and between roles and objects. In each model an object is the root of a hierarchy and roles comprise the other nodes. Those relationships are depicted in Figure 4.6, where the object *Person* can play a variety of roles. Role definitions as specialisations or generalisations of behaviour are useful in the sense that they describe different facets of object behaviour. However, to the author's knowledge they lack systematic and rigorous methods and that would allow a large number of roles to be allocated to an object in a partially automated manner.

Roles as *separate instances of behaviour joined to an object*. These definitions concern representations of behaviour that cut across the objects [108]. A common name for such representations is *aspects* [2, 111]. Examples of such behaviours provided in the literature include synchronisation, exception handling, monitoring and many others. For instance, many objects can demonstrate the same exception handling behaviour and hence exception handling can be considered an aspect. The programming paradigm that adds extra language features to object oriented programming to support handling of aspects is called Aspect Oriented Programming (AOP). Research in AOP has produced language constructs and compilers (called Aspect Weavers) that can take standard class definitions and augment them with appropriate

aspect definitions (programs) to formulate a unified and executable program. Aspects can be represented by appropriate roles. The view in this work is that such roles should be allocated to agents based on a formal model of role dependency relationships. To this end, a similar view is followed in this thesis as well.

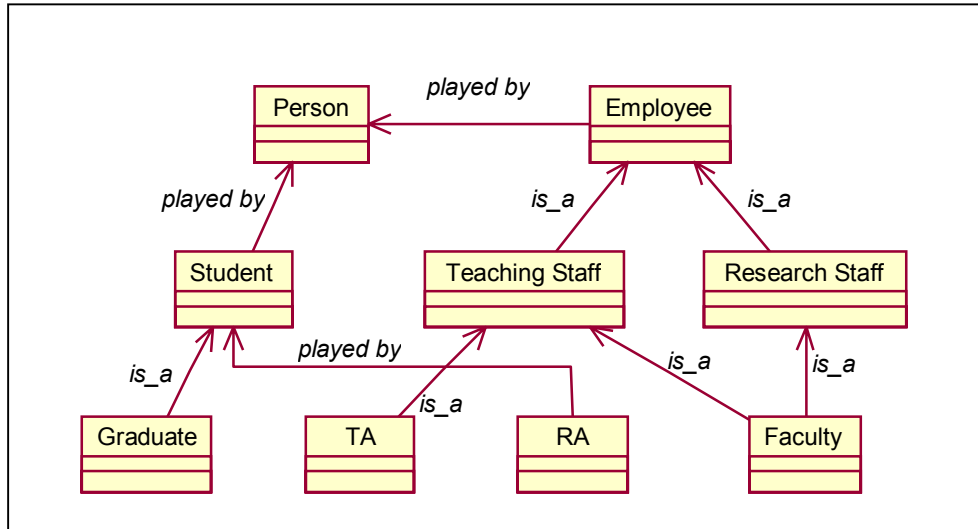


Figure 4.6: Object–Role relationships (Wong 1997)

Roles as positions filled by objects. Roles have also been used to describe sets of objects that occupy the same position in a reoccurring structure of objects [2, 108, 188]. When a number of roles are part of a reoccurring structure and represent collective behaviour based on interaction of the objects occupying the positions the roles represent, then they constitute a *role model* [2].

This view of roles facilitates separation of concern and describes the static and dynamic properties of a number of entities in a single, coherent model [108]. The main difference between roles and classes is that classes describe sets of objects that exhibit common characteristics. Classes stipulate the capabilities of the objects, while the notion of role focuses on the position and responsibilities of an object within an overall structure or system in common with the organisational and social (role theoretic) views of role concept described in Section 4.2. Representing positions and associated behaviours with roles is followed in this thesis as well.

4.3.3.2 Role Properties

Roles are a powerful abstraction and reuse construct which given an appropriate formal basis can reduce the complexity of agent organisation design. Roles are associated with a number of characteristics making it possible to represent reoccurring complex behaviour at different levels of abstraction, as outlined below.

In each role definition, role characteristics represent different facets of social behaviour. For example, according to Kendal [108], each role has a set of *responsibilities* within a role model. A role also has *collaborators* that are other roles that it interacts with. The services and

activities are accessible through an external interface. Usually there is a distinct interface for each collaboration path between two interacting entities. Object roles can also be associated with many other characteristics and a comprehensive review is given in [109].

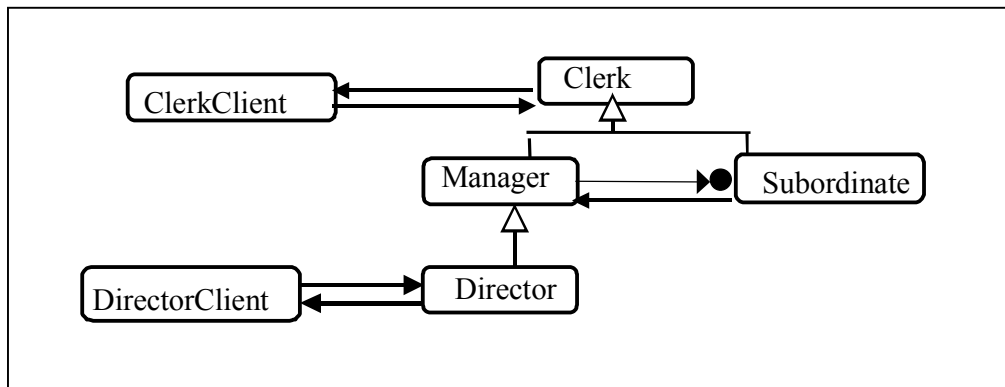


Figure 4.7: The Bureaucracy pattern represented as a role model (Richle 1997)

Along the lines of role definitions in sociology, a role in software engineering is only defined in the context of a role model. Furthermore, role assignment is generally dynamic, which means that an entity can play many roles in its lifetime, and different entities can play the same role during the course of a given application. In addition, roles can be *specialised* and *synthesised* or *composed*. Role models are instantiated in applications where software entities play the roles.

Role models can be used to document object-oriented design patterns [2, 116, 167]. A sample role model representing a design pattern named *Bureaucracy* is depicted in Figure 4.7. This pattern reflects the architecture of actual bureaucracies where there is a long chain of responsibility, a multilevel hierarchical organisation and centralized control. The role model representation is done using a non-standardised, yet common in many role modelling approaches [2, 108, 168], notation. The Bureaucracy role model includes six roles, shown with rounded boxes: *ClerkClient*, *Clerk*, *Manager*, *Subordinate*, *DirectorClient* and *Director*. The arrows between the roles indicate collaboration and the arrow direction depicts message flow. A filled circle indicates that more than one entity can play the role at a time. For example, there can be many entities playing the *Subordinate* role. In addition, a role can be specialised. Specialisation can be represented with a triangle, in the same way as class inheritance is represented in UML. For example, in Figure 4.7 it can be seen that both *Manager* and *Subordinate* specialize *Clerk*.

Apart from graphical notations for representing role models, systematic documentations of role model characteristics have been proposed. An example is the Role Responsibility Collaborators (RRC) cards, which are similar to the textual representations of use cases in object-oriented modelling [36]. A part of an RRC card for the Bureaucracy role model is shown in Figure 4.8.

Role Model: Bureaucracy	
Role: Director Client, Clerk Client	
Responsibility:	Collaborators:
request and obtain services	> Director, Client

Role Model: Bureaucracy	
Role: Director	
Responsibility:	Collaborators:
perform high level representation and management	> Subordinates, Client

Figure 4.8: Sample RRC card for the Bureaucracy pattern (Kendal 1999)

4.3.3.3 Role Relationships, Synthesis and Synergy

Roles and role models are considered as first class constructs [2, 108, 116, 168]. Therefore, new role models can be derived from existing ones using appropriate operations. In particular, roles and role models can be specialised, aggregated and synthesised to form new roles and role models. For example, role model synthesis occurs when a number of role models are instantiated at the same time and software entities must be assigned a number of roles to play. When synthesised, roles and role models may constrain each other. There is a consensus in the role modelling community that role models can be synthesised from existing ones in the following ways [2, 108, 116, 163, 166]:

Specialisation: A new role or role model may be derived (specialised) from one or more base models. In this case, the derived role must be able to play the base roles. In this sense, role specialisation is similar to multiple inheritance.

Aggregation: One role or role model may be an aggregate of other roles or role models. In that case, the behaviour the new role or role model represents is exactly the same as the overall behaviour represented by the original roles or role models.

Composition: Roles and role models may combine *synergistically* where the whole is more than the sum of its parts. Synergy is important in composite patterns and frameworks [108, 166]. Kendal [109] discusses how the *Bureaucracy* pattern, depicted in Figure 4.7, can be constructed synergistically out of four other patterns: *Composite*, *Mediator*, *Observer*, and *Chain of Responsibility* patterns descriptions of which can be found in [167].

A simple combination of the four design patterns initially results in sixteen roles for the Bureaucracy pattern, which are the following:

- Chain of Responsibility pattern: *Handler Client*, *Handler*, *Successor*, *Predecessor*, *Tail*, *Tail Client*.
- Mediator pattern: *Mediator*, *Colleague*

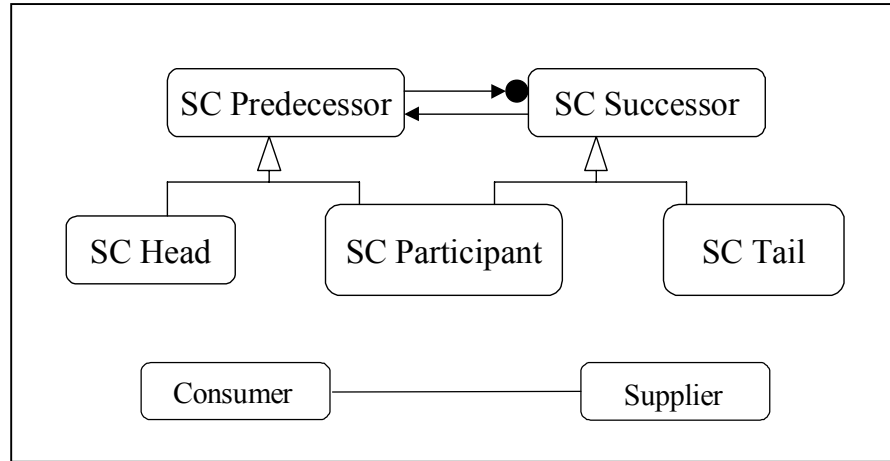


Figure 4.9: A high level view of the supply chain management role model (Kendal 1999)

- Observer pattern: *Observer, Subject*
- Composite pattern: *NodeClient, Node, Parent, Child, Root, Root Client*

However, there are in fact only six roles because the resulting compound pattern is more than a “sum” of the individual patterns. In other words, *aggregation* cannot be applied to those four patterns. This is because there are only certain valid combinations of the original roles and in addition some roles are *merged* to form completely new roles. The six resulting roles in the Bureaucracy pattern are the *Director, Director Client, Manager, Subordinate, Clerk and Clerk Client*. The role synergy occurs because, in the Bureaucracy pattern, the same entity must play more than one role. More examples of role synergy when role models describing design patterns are synthesised can be found in [109, 167].

In role composition, roles may constrain each other. Two roles may imply each other, meaning that a single entity must play both roles. Alternatively, two roles may prohibit each other; this means that the same entity can never play both roles. Examples can be taken from the *Supply Chain* role model (Figure 4.9) discussed in [109]. The same entity cannot play the roles *Supply Chain Head* (first element) and *Supply Chain Tail* (last element) since these two roles prohibit each other. However, an entity that plays the *Supply Chain Head* must also be a *Consumer*, so the *Supply Chain Head* role implies the *Consumer* role. Likewise, the *Supply Chain Tail* role implies a *Supplier* role. Such inter-role relations are formally described in the role algebra discussed in Section 5.2.7.

4.3.4 Roles in ABS Modelling

A number of approaches have used roles to represent behaviour of ABSs. These approaches extend the conventional role definitions to model the additional sophistication of the agent behaviour. The emphasis is on modelling *goal-based interactions*, and *organisational settings*.

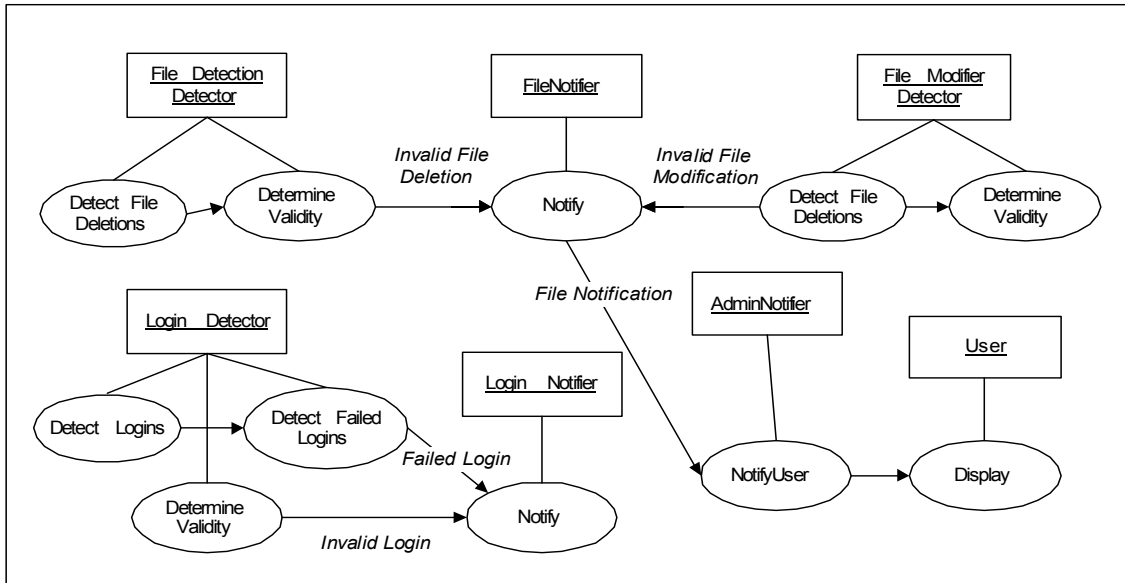


Figure 4.10: An example MASE role model (DeLoach et., al. 2001)

4.3.4.1 Modelling Goal-Based Interactions Using Roles

In ABS research, roles have been typically used to represent behaviour of interacting parties. Such a representation typically includes the protocols followed and the tasks carried out when some interaction takes place. This view of role modelling is followed in this thesis as well.

A typical example where roles are used in this manner is the *Multi-Agent System Engineering (MASE)* methodology proposed by DeLoach et al [49]. In MASE roles are used to represent the behaviour of the participants of an interaction process and correspond to the goals of the participants in that particular process,

Roles in MASE can carry out *tasks* which are associated with *task communication protocols*. A task communication protocol is the protocol followed by a role when interacting to carry out a particular task. Roles are assumed that are able to carry out tasks concurrently and hence to be able to communicate with other roles using more than one communication protocol at the same time.

Role definitions are captured in MASE using *role models*. Role models can be represented graphically by the notation depicted in Figure 4.10, which includes information on roles, tasks and communication protocols.

The same philosophy of roles mainly representing goal-oriented interaction has been adopted in many ABS modelling approaches, for example [12, 49, 113, 150]. In this thesis, roles are used to represent interacting parties as well. In addition, the view in this work is that roles can represent all types of pragmatic behaviour, for example resource consumption and monitoring, access privileges and social relations. This view of roles is described in Section 5.2.

4.3.4.2 Modelling Organisational Settings Using Roles

Roles have been used in ABS research to describe organisational settings [64, 73, 221]. This is done by using roles to represent social positions and appropriate role interactions to represent organisational relationships. Representing organisational settings in this way allows reoccurring organisational settings (organisational patterns) to be reused. However, this approach cannot be used to directly represent complex organisational rules. This approach to representing organisational settings is adopted in the work presented here.

Role-based organisation modelling aims to explicitly model the organisational relationships between the agents of an agent organisation and their fundamental and recurrent patterns. For this purpose, the notions of *role*, *interaction* and *organisational group* (or *sub-organization* [96] or *organisational structure* [62] or *society* [150] for some authors) are used [63].

In this view a *role* is considered as an abstraction of recurrent social behaviour linked to a status or a position in a society and interacting with other roles. As a result agents playing those roles are committed to specific interaction protocols with other agents and their environment. The notion of role is independent of any particular agent, an agent playing several roles and a role being played by several agents if needed.

Role interactions define the relationships linking the roles to each other. In this way various organisational relationships, such as authority relationships like “*managed by*”, can be represented by appropriate interactions between suitable roles. For example, an organisation model including a “*master role*” and “*slave roles*” — where the former is in charge of assigning work to the latter and of load balancing their activities — implicitly defines a hierarchical organisational structure and a bureaucratic management regime.

Finally, an organisational group is a set of roles and interactions between these roles representing a common context and rationale [63]. In particular, an organisational group describes a *topology* of roles and a *control regime* describing organisational relationships realised by patterns of role interaction. [220]. An example of an organisational group from human activity systems is health and safety groups. When needed, for example when a fire alarm exercise takes place, all members of a health and safety group are expected to play particular health safety roles, such as *Rescue_Team_Manager*, which are normally irrelevant to their every day duties in the human organisation.

When defining an organisation, it is necessary to specify *organisational rules* [221]. Organisational rules refer to various generic constraints, which members of an organisation have to respect. In particular, organisational rules can be [220]: (1) implicit rules moderating the interactions of all members, which are defined by generic social conventions. For example, a clerk cannot contradict or ignore the commands of his manager; and (2) company specific

behavioural constraints on how different roles can be played within different parts of the company. For example, a clerk belonging to department A cannot assume privileges for those that are members of department B. In both examples, such global constraints cannot be expressed in terms of individual roles or individual interaction protocols. Therefore, it is argued that the only way to achieve representation of such constraints is by explicit constructs concerning the whole organisation [221].

Organisational settings can be reused in a manner similar to design patterns used in software engineering. In such cases they can be referred to by the term *organisational patterns* [220]. The main difference between organisational and design patterns is that the former refer to commonly used organisational structures in ABSs. Although not currently available, it is envisaged that catalogues of organisational patterns where designers will be able to recognise in their MAS the presence of known patterns and re-use definitions from the catalogue will soon be published [212].

In this thesis, the view also is that organisational settings can be satisfactorily represented by appropriate roles and their interactions. In addition, the above ideas regarding using roles to represent organisational patterns are promising. However, existing approaches lack specific methods and techniques for incorporating organisational rules and organisational patterns in the design of ABSs [221]. The method for role-based ABS design proposed in Chapter 5 builds upon these ideas and contributes the formal base necessary for considering organisational settings and rules in the design in a systematic manner.

4.4 Using Roles for the Design of ABSs

Roles have been used in modelling systems ranging from social behaviour and information systems. In this section, the role modelling approaches described in previous sections are compared and their strengths and weaknesses are highlighted. Based on the comparative results, directions that need to be followed to use role modelling for the design of ABSs are highlighted.

4.4.1 Comparison of Role Modelling Approaches

The discussion in this chapter has revealed a number of strengths and weaknesses that are pertinent to the use of roles for designing ABSs. These are summarised in this section.

Most approaches consider roles as representations of simple, normative behaviours with the exception of those from the areas of sociology that use roles to represent sophisticated social behaviours. Regardless of the complexity and type of represented behaviour, most approaches define roles within some context and associate them with a number of duties/responsibilities that need to be fulfilled, and with a number of privileges/capabilities that can be used.

Fulfilment of responsibilities requires carrying out some tasks or functions, possibly via the use of the privileges that roles have.

Role modelling approach	Strengths	Weaknesses
Sociology Role Theory	1. Based on a comprehensive theory, it describes social role characteristics, role creation, role assignment and role dependency relations	1. It lacks adequate formalisation for implementation of roles 2. It lacks adequate formalisation for automated role allocation considering dependency relations 3. It considers a superset of normative behaviour
Business Systems Modelling	1. It involves simple and intuitive role definitions 2. There are reported implementations of role specifications	1. It has no systematic approach for identifying roles. 2. Context definitions are too high grain. 3. It lacks adequate formalisation for automated role allocation considering dependency relations
Role-Based Access Control	1. It provides adequate formalisation for automated role allocation considering dependency relations 2. There are reported implementations of role specifications	1. It does not model all facets of agent behaviour 2. It has no systematic approach for identifying roles.
Object Oriented Software Engineering	1. Provides adequate formalisation for automated role allocation 2. There are reported implementations of role specifications	1. It does not model all facets of agent behaviour 2. It lacks adequate formalisation for automated role allocation considering dependency relations
Agent-based Systems Modelling	1. Models all facets of agent behaviour, i.e. both functional and organisational behaviour	1. It lacks adequate formalisation for automated role allocation considering dependency relations

Table 4.1: Strengths and weaknesses of role modelling approaches

Role theory systematically describes how roles are identified (created) in a social system, how their tasks or functions are determined and how they are assigned to persons that perform them. Therefore, the ideas proposed in role theory for role identification have been adopted in role modelling approaches in the area of information systems engineering [112]. However, there are still ambiguities as far as it concerns assigning roles to actors. Role modelling approaches from information and business systems engineering do not provide systematic methods for allocating roles to actors.

Assigning roles to actors is highly related to possible role dependency relations. Most approaches acknowledge that role dependency relations should be taken into account and discuss the impact of possible inconsistencies to role assignment. However, apart from a few

exceptions [127], role dependency relations are not described in a rigorous manner, which could be used in role modelling for ABS design.

The strengths and weaknesses of the approaches examined in this chapter are summarised in Table 4.1. Role theoretic approaches provide comprehensive support for role identification and assignment of roles to persons, whilst approaches originating from information systems engineering are geared towards computer-based systems and are thus much more rigid. The major weaknesses identified in some role modelling approaches are that the role identification process is not clearly specified and the formalisation of role dependency relations is not sufficient to allow for automating the role assignment procedure to a certain extent. In this thesis, an effort is made to adopt the strengths and avoid the weaknesses of current modelling approaches. This is discussed further in Section 5.2.

4.4.2 Formalising Role Dependency Relations

The majority of the role modelling approaches examined in this chapter consider that various relations may exist between roles. For example, an examiner cannot be a candidate at the same time, and therefore appointing these roles to the same person at the same time results in inconsistency. Such relations are informally taken into account in some role modelling approaches, for example in the context of object-oriented software engineering (Section 4.3.3). However, there is a consensus that to be able to use role dependency relations to design role-based software, they need to be formally described in a rigorous manner [16, 108, 167].

The larger body of work on role dependency relations exists in sociology. It considers sophisticated dependencies between and across multiple roles. Designing ABSs in a practical manner requires considering only a sub-set of these inter-role relationships, those concerning how the roles participating in the relationship can be allocated to the same actor/agent. Referring back to role relationship zones discussed in Section 4.2.1.2, this thesis is interested in formalising those relationships of roles assigned to the same agent, which are situated in the *institutional* and *legal* zones (Figure 4.1). Any relationships within the *subjective*, *inter-subjective* and cultural zones will be disregarded. Formalisation of role dependencies is further discussed in Section 5.2.7.

4.5 Summary

In this chapter, the foundations of a role-based approach for the design of ABSs have been established. To provide the basis for an appropriate definition of the role concept to be used for designing agent organisations, the use of role concepts in the areas of sociology, business systems modelling, software engineering and ABSs has been reviewed. Furthermore, their

strengths and weaknesses have been highlighted, together with issues that need to be considered in role modelling for ABS design.

Roles in software engineering have been used to represent various concepts including named places in conceptual relationships, specialisations/generalisations, separate instances joined to an object and positions filled by objects. Furthermore, roles have interesting properties transferred from sociology. For example, roles can be aggregated, specialised and synthesised in various ways. However, no comprehensive methods for assigning roles to objects currently exist primarily due to lack of adequate formalisation of role dependency relations and other relevant constraints.

Role definitions in business systems and software engineering are not detailed enough to represent intelligent agent behaviour. This is addressed in many role-modelling approaches specifically targeting ABSs. However, no adequate formalisation of role dependency relations is provided and this impedes the automatic assignment of roles to agents.

In this thesis, role modelling is used as the basis of a method for ABS design, which addresses the open issues raised in Chapter 2. The method is based on the automatic assignment of roles to agents whilst observing constraints based on role dependency relations and on role characteristics. The method is described in detail in Chapter 5.

Chapter 5

The RAMASD Method

This chapter builds on the role modelling foundations defined in the previous chapter and proposes an ABS Design method called RAMASD (Role Algebraic Multi-Agent System Design). RAMASD simplifies ABS design because it allows designers to operate at high levels of abstraction, such as role models, whilst automating the allocation of roles to agents. This is enabled by the main innovation of this thesis, a formal model of role relationships termed role algebra.

5.1 Using Role Modelling and Synthesis for ABS Design

Chapter 3 has described a number of open issues in ABS design. A consensus regarding the most appropriate way to address those issues has not been reached yet. This is due to two main obstacles that ABS design methods have to overcome: selecting appropriate modelling abstractions and following a suitable design process. The ABS design method proposed here addresses these issues by using roles as modelling abstractions, by formalising relations among roles as far as it concerns allocation of roles to agents and by applying the synthesis concept to the design process.

In this chapter, a novel method, RAMASD, is proposed. RAMASD aims to provide effective solutions to these problems and address the issues raised in Chapter 3. The ABS design problem in RAMASD is viewed as that of allocating roles to agents. RAMASD uses role modelling and it is based on role theory. The RAMASD role modelling approach is complete in terms of the definition of completeness of role modelling approaches given in Section 4.1. The RAMASD design process follows the principles of *synthesis*. Synthesis involves the construction of sub-solutions for loosely coupled sub-problems and the integration of these sub-solutions into a complete solution. Furthermore, RAMASD considers collective behaviour and organisational settings as first class design constructs and it involves automatic consideration of design heuristics and non-functional aspects in design. The main innovation of RAMASD is the role algebra, a formal model of role relations that provides the basis for rigorous and semi-automatic assignment of roles to agents.

The role algebra, together with details of the role modelling used in RAMASD, is described in Section 5.2. The role modelling choices in RAMASD are often linked to the underlying process of ABS design, which uses the principles of synthesis. The principles of synthesis-based design

process are therefore described in Section 5.3. These principles are then fused with role modelling into the RAMASD design process, described in Sections 5.4. The innovative features of RAMASD and its compatibility with existing methodologies are discussed in Sections 5.5 and 5.6, respectively. Finally, the chapter is summarised in Section 5.7.

5.2 Role modelling in RAMASD

In Chapter 4, various role role-modelling approaches have been discussed. The discussion suggested that in order for a role modelling approach to be useful in ABS design, it should be complete and based on a comprehensive theory. In this section, a role-modelling approach for ABS design based on role theory is proposed. The suitability of the proposed approach for addressing the open issues raised in Chapter 2 is discussed in Section 5.5.

5.2.1 Defining Roles and Role Models

In this section, the role concept is defined considering two objectives: to represent the sophisticated behaviour of agents in a social context and to describe the characteristic properties of that behaviour so that they can be realised in software implementations. The notion of role model is used to represent a number of roles interacting for the needs of a common activity.

5.2.1.1 Role Characteristics

Following [108], a role is defined as a *position* in an ABS associated with a set of *characteristics*. Along the lines of role theory [16], roles describe some particular expected behaviour within some *social context*. Roles represent a pragmatic view of agent behaviours, for example an ABS is considered to include a specific number of roles at a given time, each one consuming system resources and contributing to changing the environment the ABS operates in. When an entity in a social system realises the behaviour represented by a role then it is said that the entity *plays* that role.

Role characteristics	Description
Role Model	Describes the application context in which the role is applicable.
Goals/Responsibilities	Refer to what the role aims to achieve within a particular context
Tasks	Represent specific tasks the role can carry out.
Capabilities/Privileges	Properties that enable/facilitate role behaviour.
Performance variables	Describe run-time aspects of role behaviour

Table 5.1: Role characteristics

In the approach proposed in this thesis each role is associated with five types of characteristics (Table 5.1): role model, goals/responsibilities, tasks, capabilities/privileges and performance variables.

Role models represent collections of roles and their interactions. A role model represents the collective behaviour required to carry out some activity² in the system. An agent application normally consists of more than one activity and hence it will involve more than one role model. Role models that occur frequently in some application domain are called *role interaction patterns*. Role models can be used to represent reoccurring complex behaviour based on multiple points of interaction. Therefore, they are considered as first class design constructs, that is they are considered as entities that can be instantiated and given identity.

In social systems the behaviour of social entities is affected by the goals the entity tries to achieve and by the duties the entity has within the social system [16, 108]. In role modelling, this is represented by defining roles to have various *responsibilities* or *goals* that they aim to achieve. The view in this thesis is that as roles represent behaviours in certain contexts, they are associated with specific duties that need to be carried out and with goals that need to be achieved in those contexts.

Role behaviour is externalised by carrying out certain *tasks*. Tasks correspond to actions that social entities take towards fulfilling their duties and achieving their goals. In carrying out tasks, roles normally need to interact with other roles, which are their *collaborators*. Interaction normally takes place by direct exchange of messages according to interaction protocols. It must be noted that not all roles interact with each other in a role model. In the extreme case, there may be a role model consisting of only one role interacting only with passive resources and the environment. For example, this is the case when an agent simply handles the temperature valve of a central heating unit. Such an agent will be playing only one role, that is monitoring the environment for changes in the temperature, and its only task will be to operate the valve accordingly.

Capabilities or *privileges* refer to properties that enable or facilitate a role to achieve its goals and fulfil its responsibilities. Examples of capabilities/privileges include learning, inferencing and communicating. This view is similar to the one of role theory where role functions — particular aspects of role behaviour — have characteristic effects on the social system in connection with the goals of roles. The notion of role capabilities is common in the majority of role modelling approaches discussed in Chapter 4.

² Activity in this context will represent the whole causal sequence of events and actions caused by one triggering event, and will correspond to the UML's concept of "use case".

Each role characteristic includes a set of *attributes*. Attributes represent different aspects of a characteristic property of role behaviour and can take both numeric and non-numeric values. For example, a characteristic of a role could be its capability to negotiate. The negotiation characteristic can have many attributes, including the name of negotiation strategy that is followed and maximum and minimum bid values.

In order for roles pragmatically represent behaviour in an application domain they need to model issues relevant to non-functional aspects in that domain. Therefore, the above role definition is extended to include *performance variables*. Performance variables are parameters whose value defines the run-time behaviour represented by a role. For example, if the behaviour represented by a role requires using some resource like memory, the resource capacity can be modelled by a performance variable. Performance variables can also be defined at an agent level. In that case, their value is a function of the values of the respective performance variables of all roles the agent is capable of playing. This allows us to apply design heuristics by imposing constraints on the values of the agent performance variables that must be observed when allocating roles to agents. This is illustrated in the example discussed in Section 7.2.

5.2.1.2 Properties of Roles and Role Models

Roles can be specialised in a manner similar to inheritance in object orientation. Furthermore, simple role models can be composed to form complex role models representing sophisticated behaviour. Roles are bounded to various constraints in role composition.

Roles can be extended to create specialised roles by a process called role *specialisation* or *refinement*. This view is similar to the one suggested in [2, 16, 108]. Specialised roles represent additional behaviour on top of the original role behaviour in a manner similar to inheritance in object-oriented systems. For example, in a university both *Student* and *Member_of_Staff* roles are specialisations of the *University_Member* role. The behaviours they represent have common aspects, for example, they can both borrow books from the library.

The task of merging a number of roles into a single composite role is called *role composition*. Role composition occurs when roles are allocated to agents. In role composition roles may semantically constrain each other. For example, two roles may constrain each other in such a way that a single agent cannot play both roles at the same time. Furthermore, the way that role characteristics and their attributes are merged may be bound to various constraints. For example, the resource capacity required by the composite role resulting from the merging of two roles may be less than the sum of the capacities required by the two individual roles. In this thesis, constraints among roles with respect to role composition are termed *compositional constraints*. Compositional constraints are captured in the role algebra, a formal model of role relations concerning allocation of roles to agents, which is described in Section 5.2.7.

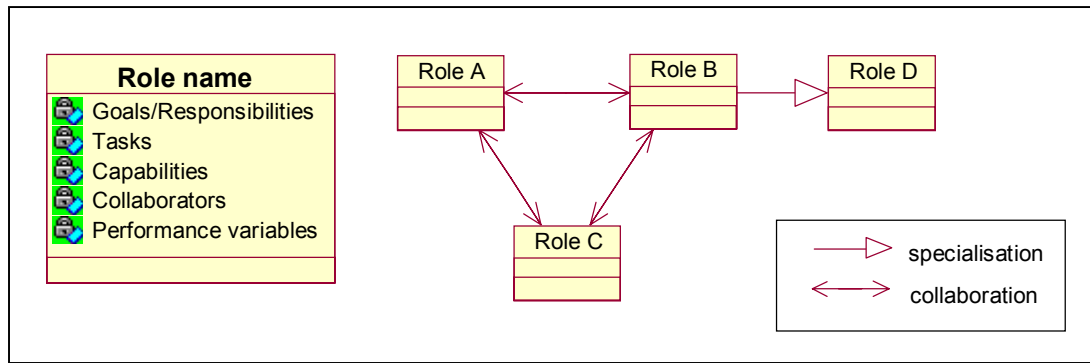


Figure 5.1: Schematic representation of a role model using UML notation

5.2.2 Representing and Using Role Models

The notation that can be used to represent role models is based on the one used to represent UML class diagrams. Each role can be represented by a rectangle similar to the one used to represent classes in UML. Optionally, role rectangles can also contain the names of role characteristics and the values of role attributes. Interacting roles are linked with association arrows whose direction represents the flow of information. Specialised roles are linked with triangled arrows in way similar to the specialisation of classes in UML. The basic UML notation used to graphically represent role models in this thesis is depicted in Figure 5.1. A more detailed notation that can be used to represent additional relations among roles is described in Section 5.2.7.3.

5.2.3 Role Model Types

Role models can be used to describe various types of behaviour, including organizational, functional and non-functional behaviour. By using compositional constraints the way that different types of behaviour is merged and allocated to agents can be specified.

RAMASD considers the following types of role models:

- *Functional role models*: They describe behaviour specific to the application domain. For example, the collective behaviour that carries out negotiation in a B2B e-commerce context can be described by a functional role model.
- *Non-functional role models*: They are used to model behaviour that implements non-functional aspects of the application. For example, to increase security of business-to-business transactions it could be required that only registered partners should be able to access the pricing information and any transactions should be carried out using a secure communications protocol. This could be modelled by representing that non-functional behaviour by the *Registered_Partner* and the *Secure_Protocol_Trader* roles and by requiring that agents should play those roles in order to be allowed to interact with other

agents in the ABS. The innovative way in which RAMASD handles non-functional aspects is further discussed in Section 5.5.4.

- *Organisational role models:* They specify organisational patterns, namely reusable organisational settings that one would like to impose on the agent system. Organisational roles further specify the agent behaviour. For example, an agent that is not capable of carrying out a task may request that its *peer agents*, which are agents at the same level in the organisational hierarchy [147], carry it out on its behalf. Organisational role models can also be used to impose organisational rules [220] and to introduce social relations [150] among agents in a multi-agent system. Those issues are further discussed in Section 5.5.3.

5.2.4 Identification of Roles in the Application Domain

Various criteria have been used for role identification both in social systems and in information system modelling. To enable the specification of a wide range of application requirements and domain solutions (patterns), this thesis accepts a wide variety of role identification criteria. Hence, roles can correspond to social positions as well as to functions and tasks that need to be carried out in the business organisation, which will be supported by the ABS.

5.2.4.1 Criteria for Role Identification

Considering the discussion about role modelling carried out in Chapter 4, the following criteria should be used for agent role identification:

Roles as personal behaviours: Along the lines of role theory, particular behaviours are associated with persons in the social system. For example, there should be a different role for each user of the ABS, which will be played by agents acting on her behalf. Even when such representation of humans is not explicitly included in the application requirements, personal roles can be used to represent the agent behaviour that links the human users with the rest of the ABS.

Roles as social positions: There is an agreement in social theories and business process modelling that in a social system there are certain positions that should be directly represented by roles. Therefore, in the process of identifying roles for ABS modelling any characteristic social positions should be specified first. Characteristic social positions can be identified based on knowledge of the social system, for example it has been suggested that open ABSs should include social positions relevant with law enforcement [47]. The view in this thesis is that such social positions should be represented by appropriate roles at the early stages of role modelling of the ABS behaviour.

Roles as service providers: In social systems there are certain functions (services) that are characteristic of the overall system purpose. Services correspond to self-contained operations

that can carry out a task, conduct a transaction or solve a problem. For example, in an ABS aiming to support an international business, a characteristic service that the ABS should offer is that of translating documents in many different languages. Such characteristic functions should also be identified early in the role modelling process and be represented by appropriate roles. For instance the translation service mentioned above should be represented by a suitable role before other roles are sought during the modelling process.

Roles as task carriers: There is a consensus in the role modelling approaches discussed in Chapter 4 that role behaviour can be described in terms of tasks performed. In line with this view, possible tasks that need to be carried out in the social system are identified first, for example via a task-based analysis method [7]. Subsequently, they should be assigned to appropriate roles. Task assignment is normally done based on well-known heuristics in software engineering. For example, according to the *point of interaction* heuristic tasks that require frequent exchange of information are assigned to the same role (see also [38]). Task-based role identification is a common practice in the majority of role-modelling approaches, for example in [108].

5.2.4.2 Goal-Oriented Role Identification

Kendal and Zhao [112] proposed a role identification method, which is based on goals. It begins with use cases, in a similar manner as they are used in standard object oriented software engineering. The use cases are identified and structured on the basis of goals following the technique presented in [36]. The result of the use case analysis is a *goal hierarchy tree*. Subsequently, the goal tree is refined in a manner similar to the one applied to class inheritance trees in object-oriented programming so that repeated goals do not appear in the tree. Finally, existing role interaction patterns are examined and goals are matched with roles where appropriate. The remaining goals are assigned to new roles as role responsibilities based on generic heuristics, for example high coupling and low cohesion.

A small extension to the above role identification method is to combine it with the role identification criteria introduced in Section 5.2.4.1. This is illustrated in the example described in Section 5.2.4.3. The phases of the amended goal identification method are then the following (Figure 5.2):

1. *Capture System Goals:* Capturing system goals begins by extracting scenarios from the requirements specification, user stories, or any available source [123]. Goal statements for each scenario are ascertained by posing the question, “What is the objective of this scenario?” Goals are identified by determining the purpose of each scenario. For example, one such goal for an open agent based supporting e-business could be to

monitor and report any confidential information security violations by agents arriving from other locations.

2. *Create Goal cases*: In order to record and track the relationships between scenarios and goals, scenarios are consolidated to use cases according to known use-case management techniques [36]. Each use case must be related to a particular goal and, therefore, it is called a *goal case*. At this stage, if more than one scenario correspond to the same goal, each scenario and its related goal represent a distinct goal case.
3. *Create Goal case tree*: There is a consensus in requirements analysis literature that there are different types of scenarios and goals, as well as hierarchical and other relationships between them [123]. Therefore, the goal cases identified in the previous phase are now structured and classified into three classes: *main discourse*, *subordinates*, and *extensions*. This can be represented by a goal case tree, where the main goal case is the root and other goal cases are subordinates and extensions of it.
4. *Refine Goal tree*: To avoid redundancy and duplication repeated goals need to be removed. This can be done by promoting redundant goals and actions to a high level tree node, and utilising inheritance to bring the common factors into subordinate nodes. Following the notation introduced in Kendal and Zhao [112], subordinates within the main discourse can be indicated with a hierarchical outline format; extensions are marked with a letter suffix.

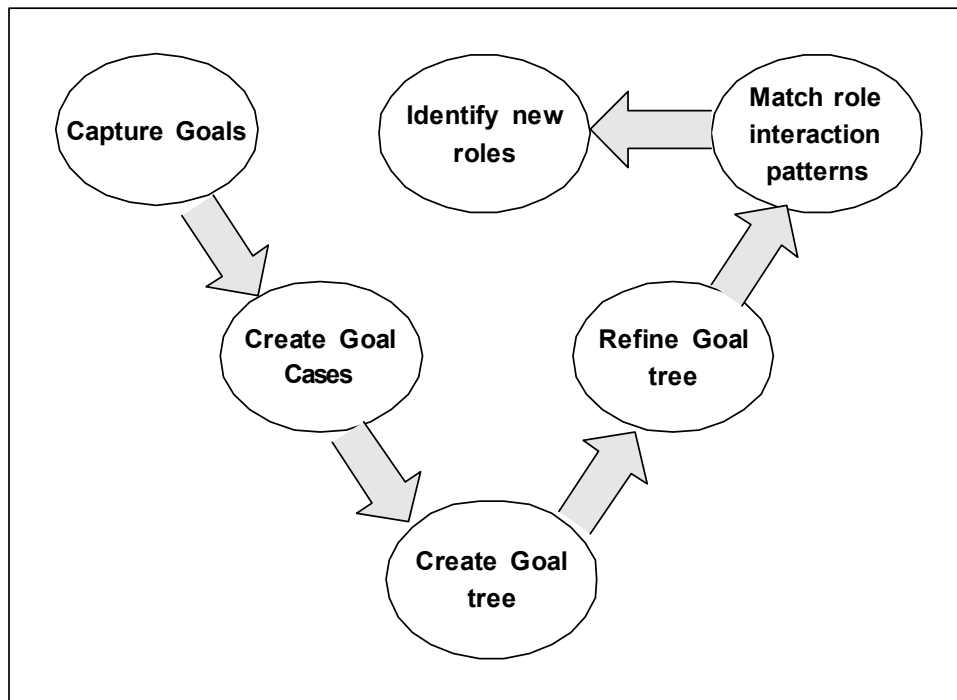


Figure 5.2: The phases of a goal-oriented role identification method

5. *Match role interaction patterns:* If a documented role interaction pattern, for example one found in the role model catalogue maintained at BT labs [109], can be reused in this application domain then the responsibilities of the role pattern roles are matched with appropriate goals from the goal tree. Those goals are then marked as ‘assigned to roles’.
6. *Identify new roles:* When assigning goals to roles from existing role interaction patterns are not applicable. Goals can be assigned to roles according to the criteria suggested in Section 5.2.4.1. In particular, new role identification consists of four steps:
 - i. *Introduce a new role for each type of user in the system.* For example, the goals pertinent to the behaviour of the system administrators in an e-business system should be represented as responsibilities of the e-business system administrator role. This practice makes the modelling of user-related system behaviour intuitively clear. Furthermore, it is in accordance with the common view that the units of analysis used to represent the problem should be semantically aligned with the constructs used in the solution [96]. Hence, since users are part of the resulting system it is necessary that certain roles should closely represent their behaviour.
 - ii. *Introduce a new role for each social position in the system.* For example, in an e-business system some sort of legal authority could be required. That legal authority should be able to take legal action (open an appropriate case) when required. Such an authority should be represented by a separate role in the system assigned with appropriate law enforcement responsibilities.
 - iii. *Introduce a new role for each distinct service (function) in the system.* A service in this context is considered a system operation (function), which can carry out tasks and solve problems but, in contrast to tasks, it does not have a specific notion of completion. It only involves the notion of interruption, which happens when the service is not provided any more. An example of such a role in an e-business security system could be the role that regularly monitors currency fluctuations and modifies product prices accordingly. According to the above definition, the behaviour of this role represents a service since it may carry out many price updates but it does not ever complete.
 - iv. *Introduce a role for each related group of tasks in the system.* Goals are achieved by tasks. Tasks can be either primitive tasks or composite tasks consisting of subtasks. At this step the tasks corresponding to the remaining goals of the goal tree are grouped and assigned to appropriate roles following heuristic guidelines. For example, each role should have high cohesion and coupling or

interdependencies across roles should be minimized. Some interdependencies will, of course, be required, these become the collaborations in the role model.

This goal-based role identification method is used in the examples throughout this thesis.

5.2.4.3 Role Identification for an e-Business Security System

The above role identification method is demonstrated in an example involving an e-business security system. In this simplistic example, the system is required to monitor security violations in an e-business system, to notify a system administrator and automatically take legal action against the intruders. For simplicity, only illegal resource accesses and system file intrusions are considered as system violations. For the needs of the example, system requirements are informally the following:

- The system is responsible for dealing with host violations, in particular resource access violations and system file intrusions. The system administrator is notified of suspected or attempted intrusions.

- | | |
|---|---|
| 1. | <i>To detect and notify system administration of host violations.</i> The system is responsible for dealing with host violations. |
| 1.1 | <i>To detect and notify system administration of system file violations.</i> The system is responsible for dealing with system file intrusions. |
| 1.1.1 | <i>To determine if system files have been deleted or modified.</i> It is necessary to validate the date, time and existence of system files periodically, every few minutes. When a file is not found or a new version appears, this is a violation. |
| 1.1.2 | <i>To detect an attempt at system file violation.</i> When a user tries to modify or delete a system file, this is a violation. |
| 1.1.3 | <i>To notify system administration of system file violations.</i> The system administration needs to be notified of system file violations. |
| 1.1.3a. | <i>To ensure that system administration receives notification of a violation.</i> The system administrator may not be available to receive a notification. For example, this can be due to a network failure. |
| 1.2 | <i>To detect and notify system administration of login violations.</i> The system is responsible for dealing with login violations. |
| 1.2.1 | <i>To determine if an invalid user tries to login.</i> A user tries to login when he or she does not have a valid account. If this occurs once or twice in a short period of time, it is not a violation. Three or more attempts is a violation. |
| 1.2.2 | <i>To notify system administration of login violations.</i> The system administration needs to be notified of login violations. |
| 1.2.2a. | <i>To ensure that system administration receives notification of a violation.</i> The system administrator may not be available to receive a notification. This can be due to a network failure or the fact that the administrator is performing another task. The report needs to be stored and resent after a delay. |
| 1.3 | <i>To monitor and record system violations.</i> Monitoring should be constant and any system violations should be properly recorded. |
| 1.4 | <i>To take legal action against intruders.</i> Any security violations must automatically launch and for legal action process. |
| 1.5 | <i>To increase security measures as required.</i> Any repetitive security violations need appropriate action to be taken by the system administrator who will increase security measures. |
| Note: Goal cases 1.1.3 and 1.2.2 are duplicates of the same goal “Notify System Administrator.” The conditional extensions of 1.1.3a and 1.2.2a are also the same. These can be treated as instances of the same class, and hierarchical relationships can be represented with inheritance in the goal tree. | |

Figure 5.3: Goal cases for an e-business security protection system

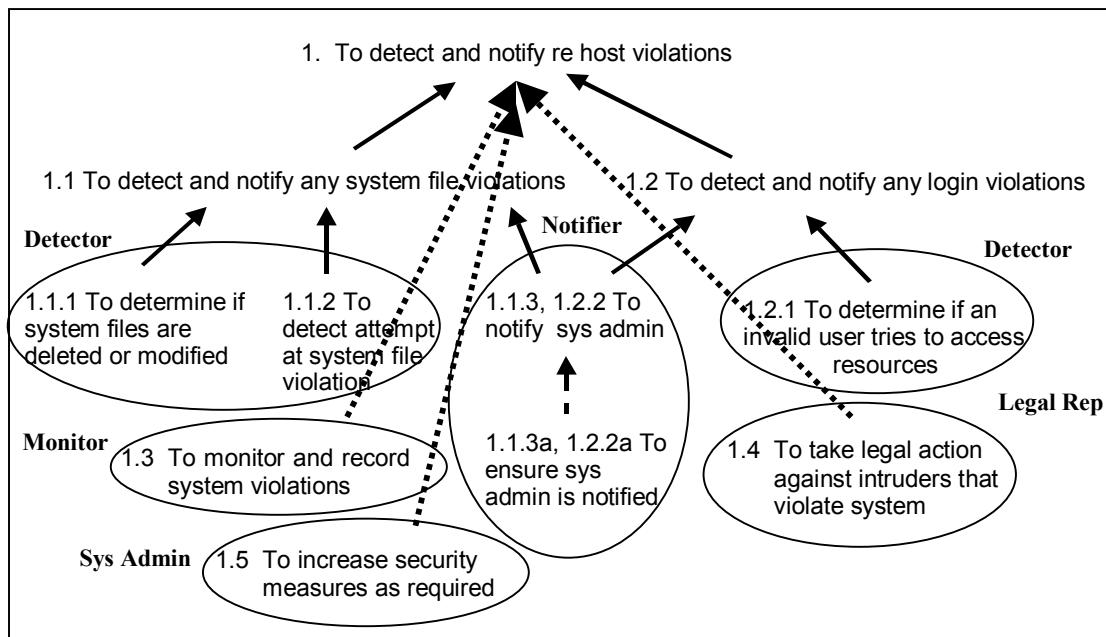


Figure 5.4: Goal hierarchy tree and role identification for an e-business security system

- It is necessary to validate the date, time and existence of system files periodically, every few minutes. When a file is not found or a new version appears or a user tries to modify or delete a system file, the system administrator needs to be notified.
- A user tries to access a resource (i.e. a database) for which he does not have appropriate privileges.

The possible goal cases of this example are documented in Figure 5.3. A sample refined goal case tree is shown in Figure 5.4. Assuming that no role interaction patterns can be reused, goals are assigned to roles in the following order:

1. The human actor involved in this system is the system administrator. Therefore, a *Sys_Admin* role is introduced. The system administrator is responsible for taking security measures once system security violations have been reported. Part of the tasks relevant to this duty of the system administrator can be automated and carried out by the ABS. This is represented in the requirements by Goal 1.5. *Sys_Admin* models the behaviour of the system administrator that is carried out by the ABS and therefore, it is naturally assigned the responsibility to achieve Goal 1.5.
2. The system also includes a social position involving taking legal action against system security violators. This corresponds to Goal 1.4. This social position is modelled with the *Legal_Rep* role which has the responsibility to achieve Goal 1.4, namely to automatically open a legal action case once a security violation has been reported.

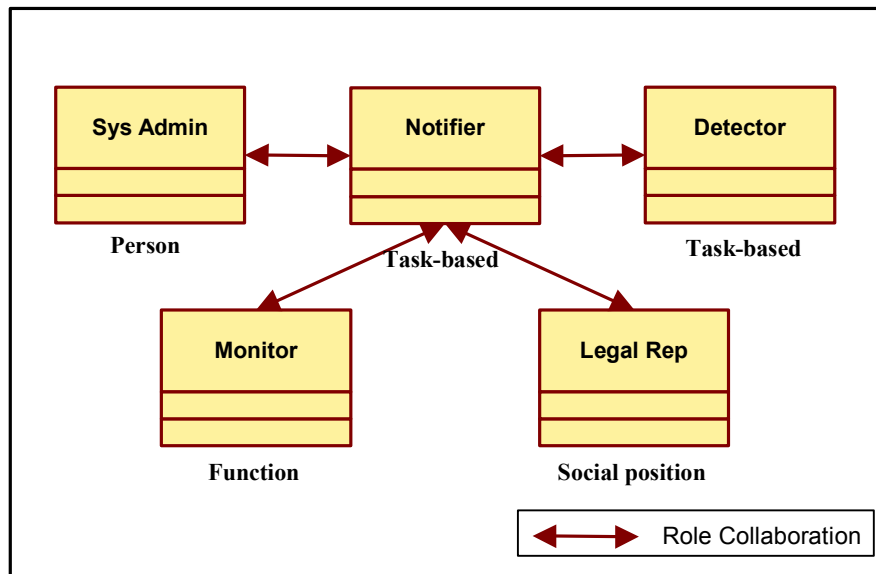


Figure 5.5: Identified roles for the e-business security system

3. To increase system protection the e-business security system constantly monitors for reports on system violations. When a system violation is reported then it is recorded for analysis and future prevention. This system function corresponds to Goal 1.3 and it is represented by the *Monitor* role.
4. The remaining goals correspond to tasks that may need to be carried out in the system. Goals 1.1.3, 1.2.2, 1.1.3a and 1.2.2a are similar in that they all involve contacting the system administrator and providing system security violation information. Therefore, they are all represented as responsibilities of the same role, the *Notifier* role. In the same way, Goals 1.1.1, 1.1.2 and 1.2.1 all involve detecting some system security violation and therefore they are modelled as the responsibilities of the *Detector* role.

The resulting role model comprising the identified roles is depicted graphically in Figure 5.5.

5.2.5 Management of the Role Modelling Process

Role modelling is considered to be an informal process carried out completely by human designers. Furthermore, the view in this thesis is that role modelling should definitely be completed before the design and deployment of the ABS.

As this role modelling method aims at supporting the design of ABSs, role modelling is expected to be carried out by ABS designers. It is to be carried out manually since the process of role identification is largely based on the approach followed in representing user requirements. Some degree of automation in role identification could be achieved, however, if requirements are described in a formal manner, for example in some formal specification language such as the one proposed in [80]. The criteria for role identification introduced in Section 5.2.4 could then be formally represented in the requirements specification language

constructs and the role identification process could be automated. In that case, the focus of the efforts of the designer would be the formal description of system requirements. In this thesis, informal requirements specification methods are assumed. For example, in Section 5.2.4.2 application requirements were specified using textual descriptions and use-cases. Therefore, in this thesis role identification is considered a completely informal step in the overall design process.

In the proposed role modelling method, role modelling should be completed before the ABS design phase. The premise for this is that, although the ABS behaviour may be dynamic, the application requirements are fixed and hence, appropriate role modelling can be completed before design. This is similar to static approaches to ABS engineering discussed in Chapter 2 and in particular to [157, 177], where the ABS behaviour is modelled, verified and evaluated before actual system deployment aiming to reduce consumption of system resources due to reorganisation and increase system stability. Considering dynamically changing application requirements is an issue that is increasingly reported as important in engineering agent-based applications. For example, to engineer ABSs to support dynamically evolving business [19]. This issue is the basis of an interesting direction of possible future research and it is further discussed in Section 9.4.

5.2.6 Consistency of Role-Based Specifications

As discussed in Chapter 4, describing agent behaviour in terms of roles that agents play can lead to a number of inconsistencies because roles can be related with each other in several ways. In the proposed role modelling method inconsistencies that may be caused by assigning roles to agents are systematically modelled considering formalised relations among roles.

The type of inconsistency in role specification considered here is due to incompatibilities between particular roles played by the same agent, which result in problems in role allocation as discussed in Section 4.2.4. For example, in most civilised societies a policeman cannot also be a judge. These two roles cannot coexist. Another example is when an academic is also a private consultant and this may lead to conflict of interest and time. The modelling approach proposed in this thesis is able to capture such cases. It formalises a set of basic relations among roles using a formal model discussed below.

5.2.7 Rigorous Role Assignment Using Role Algebra

Role relations can be instrumental in describing agent behaviour in a systematic manner. In particular, a formal definition of role relations as far as it concerns assignment of roles to agents can be used for automating the role assignment process whilst avoiding inconsistencies in specifying agent behaviour.

Using role theory [15] and case studies of human activity systems, for example [187], six basic role relations have been identified. These role relations are formally defined here in a model called *role algebra*. Using the six relations from the role algebra, constraints driving the assignment of roles to agents can be specified to serve as an input to a semi-automated agent design process. In this section, the role relations are formally defined and their meaning is informally described by intuitive examples. Subsequently, a formal description of the semantics of the role relations is given using a two-sorted algebra.

5.2.7.1 Relations in the Role Algebra

Let R be a set of roles. For any $r_1, r_2 \in R$, the following binary relationships may hold:

1. **Equals (eq)** — This means that r_1 and r_2 describe exactly the same behaviour. For example, the terms *Advisor* and *Supervisor* can be used to refer to people supervising PhD students. When two roles are equal, an agent playing the first role also plays the second at the same time. The relation $Equals \subseteq R \times R$ is an equivalence relation since it is reflexive, symmetric and transitive:
 - a) $\forall r : R \ (r \text{ eq } r)$
 - b) $\forall (r_1, r_2) : R \times R \ (r_1 \text{ eq } r_2 \Rightarrow r_2 \text{ eq } r_1)$
 - c) $\forall (r_1, r_2, r_3) : R \times R \times R \ ((r_1 \text{ eq } r_2) \wedge (r_2 \text{ eq } r_3) \Rightarrow (r_1 \text{ eq } r_3))$
2. **Excludes (not)** — This means that r_1 and r_2 cannot be assigned to the same agent simultaneously. For example, in a conference reviewing agent system, an agent should not be playing the roles of *Paper_Author* and *Paper_Reviewer* at the same time. Furthermore, a role cannot exclude itself — if it could then no agent would ever play it. Therefore, the relation $Excludes \subseteq R \times R$ is anti-reflexive and symmetric:
 - d) $\forall r : R \ (\neg(r \text{ not } r))$
 - e) $\forall (r_1, r_2) : R \times R \ (r_1 \text{ not } r_2 \Rightarrow r_2 \text{ not } r_1)$
3. **Contains (in)** — This means that a role is a sub-case/specialisation of another role. Therefore, the behaviour the first role represents completely includes the behaviour of the second role. For example, a role representing *Manager* behaviour completely contains the behaviour of the *Employee* role. When two roles are composed such that the first contains the second, the resulting role is the first role. Therefore, the relation $Contains \subseteq R \times R$ is reflexive and transitive:
 - f) $\forall r : R \ (r \text{ in } r)$
 - g) $\forall (r_1, r_2, r_3) : R \times R \times R \ ((r_1 \text{ in } r_2) \wedge (r_2 \text{ in } r_3) \Rightarrow (r_1 \text{ in } r_3))$

4. **Requires (and)** — The *Requires* relation can be used to describe that when an agent is assigned a particular role, then it must also be assigned some other specific role as well. This is particularly applicable in cases where agents need to conform to general rules or play organisational roles. For example, in a university application context, in order for an agent to be a *Library_Borrower* it must be a *University_Member* as well. Although the behaviour of a *Library_Borrower* could be modelled as part of the behaviour of a *University_Member*, this would not be convenient since this behaviour could not be reused in other application domains where being a *Library_Borrower* is possible for everyone. Furthermore, each role requires itself. Intuitively, the roles that some role r requires are also required by all other roles that require r . Therefore, the relation $Requires \subseteq R \times R$ is reflexive, and transitive:

a) $\forall r : R (r \text{ and } r)$

b) $\forall (r_1, r_2, r_3) : R \times R \times R ((r_1 \text{ and } r_2) \wedge (r_2 \text{ and } r_3) \Rightarrow (r_1 \text{ and } r_3))$

5. **Addswith (add)** — The *Addswith* relation can be used to express that the behaviours two roles represent do not interfere in any way. For example, the *Student* and the *Football_Player* roles describe non-excluding and non-overlapping behaviours. Hence, these roles can be assigned to the same agent without any problems. The relation $Addswith \subseteq R \times R$ is reflexive and symmetric:

a) $\forall r : R (\neg(r \text{ add } r))$

b) $\forall (r_1, r_2) : R \times R ((r_1 \text{ add } r_2) \Rightarrow (r_2 \text{ add } r_1))$

6. **Mergewith (merge)** — The *Mergewith* relation can be used to express that the behaviours of two roles overlap to some extent or that different behaviour occurs when two roles are put together. For example, a *Student* can also be a *Staff_Member*. This refers to cases where PhD students start teaching before they complete their PhD. Although members of staff, these persons cannot access certain information (e.g. future exam papers) or have full staff privileges due to their student status. Also, their salaries are different. In cases like this, although the two roles can be assigned to the same agent, the characteristics of the composed role are not exactly the characteristics of the two individual roles put together. The relation $Mergewith \subseteq R \times R$ is symmetric:

a) $\forall (r_1, r_2) : R \times R ((r_1 \text{ merge } r_2) \Rightarrow (r_2 \text{ merge } r_1))$

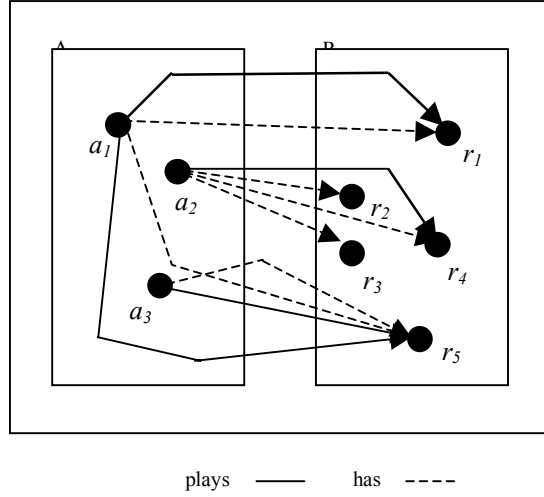


Figure 5.6: Semantics of the role algebra

5.2.7.2 Semantics of the Role Algebra

To describe the semantics of role relations an agent organization is represented by a two-sorted algebra (Figure 5.6). The algebra includes two sorts, A representing agents and R representing roles and two auxiliary relations, *Has* and *Plays* representing role allocation.

Let *Has*: $A \rightarrow R$ be a relation mapping agents to roles. The term “has” means that a role has been allocated to an agent by some role allocation procedure or tool. It is possible for an agent to have roles that do not contribute to defining the agent behaviour. For example, this happens when roles merge with other roles. For each $a \in A$, let $a.has$ be the set of roles that the agent a maps in the relation *Has*. In other words, $a.has$ denotes the relational image of the singleton $\{a\} \subseteq A$ in the relation *Has*.

Let *Plays*: $A \rightarrow R$ be a relation mapping agents to roles again. The term “plays” means that the behaviour a role represents is actively demonstrated by the agent, for example the role does not merge with other roles that are also played by the agent. For each $a \in A$, let $a.plays$ denote the set of roles that the agent a maps to in the relation *Plays*. In other words, $a.plays$ denotes the relational image of the singleton $\{a\} \subseteq A$ in the relation *Plays*.

By definition, all agents must have the roles they play:

$$\forall a : A, r : R \cdot (r \in a.plays \Rightarrow r \in a.has)$$

The meaning of the relations between roles introduced in Section 5.2.7.1 can now be described as follows:

- **Equals** — An agent has and plays equal roles at the same time.

$\forall a : A, (r_1, r_2) : R \times R \cdot (r_1 \text{ eq } r_2 \Leftrightarrow ((r_1 \in a.has \Leftrightarrow r_2 \in a.has) \wedge (r_1 \in a.plays \Leftrightarrow r_2 \in a.plays)))$

- **Excludes** — Excluded roles cannot be assigned to the same agent.

$\forall a : A, (r_1, r_2) : R \times R \cdot (r_1 \text{ not } r_2 \Leftrightarrow \neg(r_1 \in a.has \wedge r_2 \in a.has))$

- **Contains** — Contained roles must be assigned and played by the same agent as their containers.

$\forall a : A, (r_1, r_2) : R \times R \cdot (r_1 \text{ in } r_2 \Leftrightarrow ((r_2 \in a.has \Rightarrow r_1 \in a.has) \wedge (r_2 \in a.plays \Rightarrow r_1 \in a.plays)))$

- **Requires** — Required roles must be played by the same agent as the roles that require them.

$\forall a : A, (r_1, r_2) : R \times R \cdot (r_1 \text{ and } r_2 \Leftrightarrow (r_1 \in a.plays \Rightarrow r_2 \in a.plays))$

- **AddsWith** — There is no constraint in having or playing roles that add together.

$\forall a : A, (r_1, r_2) : R \times R \cdot (r_1 \text{ add } r_2 \Leftrightarrow (r_1 \in a.has \Rightarrow ((r_2 \in a.has \vee r_2 \notin a.has) \wedge (r_2 \in a.plays \vee r_2 \notin a.plays))))$















- **MergesWith** — When two roles merge only the unique role that results from their merge is played by an agent.

$\forall a : A, (r_1, r_2) : R \times R \cdot (r_1 \text{ merge } r_2 \Leftrightarrow \exists! r_3 : R \cdot ((r_1 \in a.has \wedge r_2 \in a.has) \Rightarrow (r_1 \notin a.plays \wedge r_2 \notin a.plays \wedge r_3 \in a.has)))$

For example, let us assume that roles r_2 and r_3 merge resulting to role r_4 . Based on the above semantic definition, if an agent has r_2 and r_3 then it must also have r_4 and it must not play r_2 and r_3 (the agent may or may not play r_4 depending on the relations of r_4 with the other roles the agent has). The example of a *Mergeswith* relation between roles r_2 , r_3 , and r_4 , assigned to agent a_2 , is depicted in Figure 5.6. The fact that agent a_2 has all three r_2 , r_3 , and r_4 is represented by a dotted line corresponding to the relation *Has*. The fact that agent a_2 can possibly play r_4 but it can definitely not play r_2 and r_3 , is represented by a solid line corresponding to the relation *Plays*.

Using the above semantic axioms, it is trivial to verify that the properties of role relations introduced in Section 5.2.7.1 hold.

Finally, relations between more than two roles can be defined in a similar manner. In that case, a predicate notation is more convenient to represent role relations. For example, when three roles r_1 , r_2 , and r_3 merge to r_4 this can be noted by $\text{merge}(r_1, r_2, r_3, r_4)$. It is beyond the scope of this thesis to provide formal definitions of relations among roles with arity greater than two.

notation type relation	non-interacting	interacting
addswith		
equals		
excludes		
contains		
requires		
mergeswith		
mergesto*		

*rolename can be omitted when obvious

Figure 5.7: Graphical notation for the relations of the role algebra

5.2.7.3 Graphical Representation of Role Relations

The relations of the role algebra can be represented graphically by extending the notation introduced in Section 5.2.2. To show both role interactions and role relations on the same diagram a notation for representing role relations was introduced, and when the roles are both related and interacting the linking line is amended with solid arrowheads at both ends. Since the containment relation describes role specialisation, the same graphical notation can be used for both. Furthermore, in the case of interacting roles that are simply related with the *addswith* relation, the notation is the same as the one used in Section 5.2.2 for interacting roles. The proposed notation is summarised in Figure 5.7 and it is used throughout the thesis.

5.3 Applying the Synthesis Concept to ABS Design

Synthesis is a well-known problem solving concept in traditional engineering disciplines and it is widely applied to design, for example [132, 134]. Given a clear definition of the overall problem and the possible solutions to it, synthesis employs a process of systematic selection of a solution from a number of alternatives. In ABS design, the problem is producing an ABS system satisfying the application requirements and the possible solutions are possible ABSs that can fulfil them. Therefore, it is argued in this thesis that synthesis is applicable in ABS design.

5.3.1 Synthesis in Traditional Engineering

The term ‘*synthesis*’ in engineering disciplines refers to an approach in which a problem specification is transformed to a solution by decomposing the initial problem into loosely

coupled sub-problems. The approach involves a problem-solving process in which sub-problems are independently solved and integrated into an overall solution, while various constraints within and among sub-solutions are observed [4, 174]. Problem-solving consists of searching among solution alternatives in the corresponding solution domain and selecting appropriate solutions based on explicit quality criteria. It is often useful to apply the synthesis approach iteratively at different levels of abstraction. For example, the designer may initially consider only a small number of design constraints and progressively increase this number upon obtaining satisfactory design results.

A synthesis problem solving process typically contains multiple cycles (Figure 5.8), where a *synthesis cycle* corresponds to a transition (transformation) from one synthesis state to another. A *synthesis state* can be formally defined as a tuple consisting of a *problem specification part* and a *problem solution part* [134]. The problem specification part defines the set of sub-problems that still need to be solved. The problem solution part represents the tentative design solution to a number of synthesis sub-problems. After each synthesis state transformation a sub-problem is solved and its solution is included in the solution part. Furthermore, new sub-problems can be added to the problem specification part of the new synthesis state if required. Initially, the problem solution part is empty and the problem specification part includes the initial problem requirements.

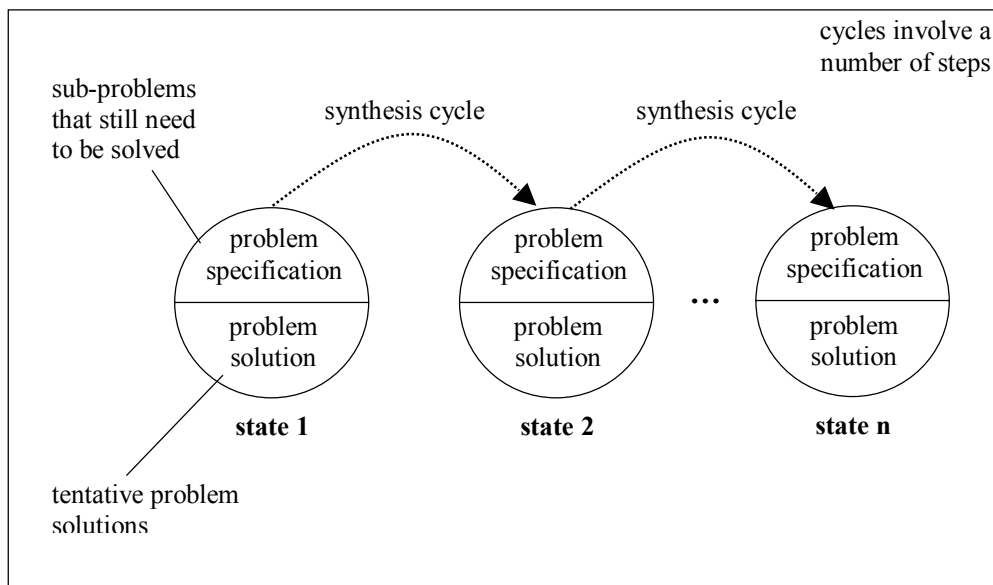


Figure 5.8: The synthesis problem solving process

Each synthesis cycle involves a number of *synthesis steps*. A synthesis step refers to a specific part of the transformation corresponding to a synthesis cycle. Each synthesis cycle includes steps for searching and selecting solutions to sub-problems based on quality criteria and for

evaluating whether the currently selected solutions are consistent with the initial problem requirements and any additional synthesis constraints.

The sequence of the synthesis cycles, results in a *terminal state* [194]. A synthesis state is terminal in either of two cases: the specification part is fully satisfied by the solution part (there is an overall solution) or neither the solution nor the specification can be modified. The former is a successful overall solution while the latter is an unsuccessful one.

The final synthesis solution must have achieved a set of *objective metrics*, while satisfying a set of *constraints*. Constraints may be imposed within and among the sub-solutions. Ideally, all objective metrics should be met to the maximum extent to provide optimal solutions. In practice, however, this is very difficult to achieve since the search of the problem solution space is an NP-complete problem [134]. Therefore, sub-optimal but consistent solutions are often considered satisfactory [43, 161]. However, even when sub-optimal solutions are sought, the search of problem solution space can still become intractable due to the large number of entities and their relations that need to be considered in large synthesis problems.

To address the above difficulties, the synthesis process is often performed iteratively at *different levels of abstraction* [161]. For example, Gajski et al. [71] propose a synthesis approach for the design of digital signal processing systems, which can be applied at four increasing levels of abstraction: circuit synthesis, logic synthesis, register-transfer synthesis and system synthesis.

There is a consensus in the literature that applying synthesis at higher levels of abstraction reduces the number of entities and relations that have to be considered, resulting to tractable search of the design solution space [71, 132, 194]. In addition, smaller numbers of entities and relations are easier to understand by humans. This facilitates evaluation of the various design alternatives by the designers in synthesis-based design. The only disadvantage is that higher-level abstractions implicitly reduce the number of possible alternatives. This is normally not a problem since the design solution space of a synthesis process performed at high abstraction level is normally large enough to be of practical use [194]. However, when this is not the case, sophisticated algorithms involving multiple search phases that combine synthesis results from different abstraction levels can be used, for example the one described in [88].

5.3.2 A Synthesis-Based Design Process Model

The synthesis concept has been applied to solve design problems in many areas of computing including hardware configurations [11], real-time software [164] and software architectures [194]. Some characteristics of the synthesis-based design approach are observable as common in all cases. Based on those common characteristics, a *generic synthesis-based design process model* can be identified including the following phases (Figure 5.9):

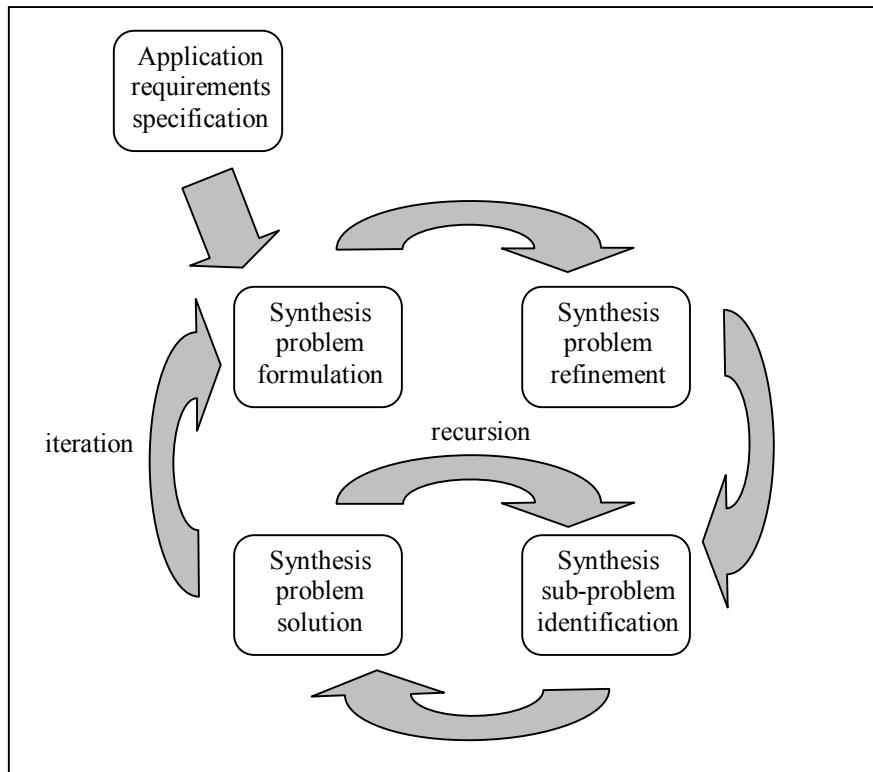


Figure 5.9: A generic synthesis-based design process model

1. *Application requirements specification*: This phase involves specifying the application requirements. This can be done using well-known requirements analysis techniques such as use-cases [94], scenarios [118] and formal specification languages [80]. Each requirements analysis approach has strengths and weaknesses. For example, use-cases provide a more precise and broader perspective of the requirements by specifying the external behaviour of the system from different user perspectives. Scenarios are instances of use-cases and they define the dynamic view and the possible evolution of the system. Finally, formal specification languages are particularly suitable for safety critical systems that need rigorous specifications. In principle, any requirements specification technique can be used. However, the requirements analysis technique used should allow grouping of requirements to individual modules [194]. This would facilitate both the formulation of the overall synthesis problem and the identification of individual sub-problems.
2. *Synthesis problem formulation*: In this phase, the overall synthesis problem is formulated and the solution domain is identified. This normally requires a formal specification of a) the design problem using mathematical formalisms [190] or a formal specification language [131], and b) any constraints arising out of the application requirements. At the design of an ATM switch [120] for example, the overall system

level architecture of the switch is described by a high-level model and all the possible configurations of different parts (the design solution domain) are specified. At this point, some objective metrics for a solution to be satisfactory are also specified. For example, in the *Formal Synthesis Hardware Design* methodology proposed in [11], the overall design goal as well as two objective metrics, the *subgoal function* and the *validation function*, are specified in this phase. These metrics are considered by the design space exploration algorithm to determine when acceptable solutions are found.

3. *Synthesis problem refinement based on the solution domain knowledge.* The synthesis problem can be further refined when the solution domain is taken into account. This allows specification of additional constraints to prevent unnecessary examination of inappropriate solution alternatives. For example, considering domain specific architectural features in ATM switch design was found to reduce the design time by 15% [120]. In software design, synthesis problem refinement should be based on knowledge of the software application domain [194]. For example, based on the standard security pattern where data access is only possible where necessary [171], the software components supporting auction participants cannot be supporting an auction coordinator at the same time since auction bidders should not be allowed access to the details of other auction bidders. Such knowledge can be used to refine synthesis-based software design problems even when the application of the particular security pattern is not explicitly included in the software requirements.
4. *Synthesis sub-problem identification:* Having specified the synthesis problem an important phase is the partitioning of the overall problem to sub-problems, which can be solved separately. For example, in the ATM switch design approach proposed in [120] the overall ATM switch design problem is partitioned to a number of sub-problems, each one corresponding to the design of a different ATM module. Each module is represented by a different VHDL process and the respective design sub-problem is solved separately. Partitioning of the problem is done in such a way so that there is clear benefit regarding the latency of the resulting designed system.

Generally, the way that the design problem is partitioned into sub-problems affects the time required to find an overall solution. Therefore, in many cases, for example in [131], the partitioning method is the starting point for an optimised design space exploration algorithm.

5. *Deriving the problem solution:* At this point the problem solving process is initiated and it is followed until it reaches a terminal state. Deriving the synthesis problem solution is intrinsically difficult since it often requires exploration of an extensively large design

space which makes the search process intractable. Therefore, developing an efficient algorithm for searching among alternatives and selecting a suitable design solution is an issue of major concern in synthesis-based design approaches [33, 58].

In many approaches the search for an appropriate overall solution involves iteration and recursion [194]. *Iteration* refers to repeating the phases of the synthesis process in order to improve the synthesis results. In particular, iteration involves a new problem definition possibly with a more satisfactory solution. For example, many synthesis-based embedded system design approaches, for instance the one described in [144], iteratively change the problem specification so that more functionality is carried out by software given that performance objectives are met. *Recursion* refers to repeating the partitioning and search phases of a synthesis process for a lower abstraction level. This involves repeatedly decomposing the overall problem into sub-problems and searching for an appropriate solution considering more problem specification details. For example, in [190] more system specification details are considered in search if the synthesis results are unsatisfactory. Generally, to obtain satisfactory solutions a number of recursions and iterations are required.

This synthesis-based design process model is the basis of the process to ABS design proposed in this thesis. This is further discussed in Section 5.4.

5.4 The RAMASD Design Process

This section introduces the RAMASD design process, which realizes the synthesis design process model described in Section 5.3.2. The process includes five phases, some of which are automated using the role algebra.

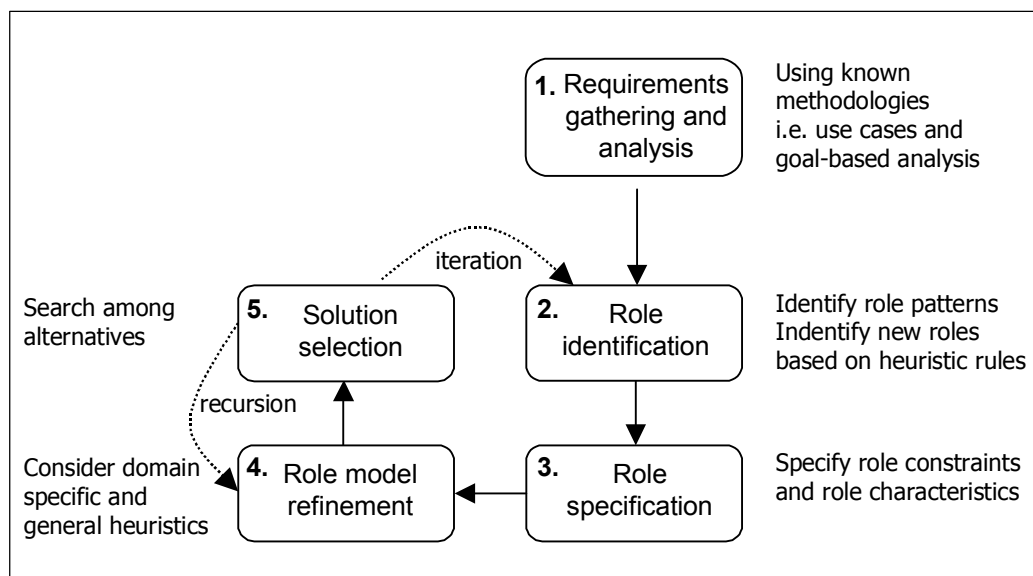


Figure 5.10: Schematic representation of the RAMASD design process

The manual and automatic steps of the semi-automatic method to role-based agent organisation design are the following (Figure 5.10):

1. *Specify application requirements*: The ABS design is initiated with the requirements analysis phase in which the basic goal is to understand the stakeholder requirements. The well-known requirements analysis techniques such as textual requirements specifications, use-cases [94] and even formal requirements specification languages [80] can be used. In all cases, however, requirements should be specified in such a way to assist role identification, which follows in the next stage.
2. *Identify roles and role models*: There are many ways to carry out role-based analysis and a systematic role modelling method was proposed in Section 5.2. Typically, role identification approaches start from use cases and for each use case identify roles and their interactions [2]. Many role interaction patterns can be used directly from existing role pattern libraries like the one documented at BT [108]. Selection or definition of appropriate role models is a manual step that must be carried out by the agent system designers.
3. *Specify relevant role characteristics and compositional constraints*: This is an automatic step since role characteristics and inter-role relations are expected to be stored in a role model library. After the designer selects existing role models, role characteristics and role compositional constraints are automatically retrieved.
4. *Refine role models*: The agent system designer is expected manually specify role characteristics and role relations for any new role models he or she defines. These new role models should be stored in the role model library. At this stage, additional characteristics of existing role models, for example performance variables, should also be specified. Furthermore, at this step various domain specific and general design heuristics are specified as constraints on the performance variables of roles and agents.
5. *Assign roles to agents*: Performing the search among various alternatives and allocating roles to agents can be done automatically. However, at this point the process may continue to a next recursion or iteration cycle depending on the results obtained. Iteration and recursion possibilities make the design of the agent organisation an interactive process where routine tasks are automated and humans carry out tasks requiring experience and creative decisions.

5.5 The Innovative Features of RAMASD

RAMASD uses role models as primary constructs for representing agent-behaviour, and this underpins the way in which it addresses the open issues of ABS design (see Chapter 2). This section presents these features of RAMASD in detail.

5.5.1 The Philosophy of the RAMASD Approach

The main concepts of RAMASD are role modelling, the role algebra and synthesis. Role models are used to represent behaviour based on multiple points of interaction; the role algebra formalises relations among roles and the synthesis concept allows for a systematic process for finding appropriate design solutions. Those concepts are instrumental in addressing the open issues raised in Chapter 2.

Role relations, as defined in role algebra, restrict the way that roles can be allocated to agents. The agent organisation design problem is thus transformed to a synthesis problem that must be solved for roles to be allocated to agent types. The problem can be constrained further by including constraints based on general design heuristics. These constraints are expressed on the performance variables of the agents. For example, the system designer should be able to define the maximum number of roles that an agent could play, or an upper limit to the resource capacity that an agent would require. Furthermore, application specific heuristics could also be specified. For example, roles requiring access to similar resources could be assigned to the same agent.

From a synthesis point of view, allocation of each individual role model to agent types constitutes a separate synthesis sub-problem. Achieving a feasible allocation of all role models thus involves merging solutions of synthesis sub problems to formulate the overall synthesis problem solution. Such a modular approach facilitates exploration of the large design solution space and enables discovery of feasible design solutions.

5.5.2 Reusing Collective Behaviour

RAMASD models collective behaviour explicitly using role models. The role algebra provides the formal underpinnings for rigorous pattern composition and hence effective pattern use.

5.5.2.1 Representing and Using Patterns of Behaviour

Patterns refer to *solutions* to documented *problems*, usually accompanied by a description of the *context*, which specifies where those solutions are applicable [41]. In RAMASD, patterns are used in the same way to refer to reoccurring behaviour in ABSs. Modelling design patterns using roles integrates both the static and dynamic aspects of the pattern in one model. Furthermore, it facilitates combining functional, non-functional and organisational behaviour.

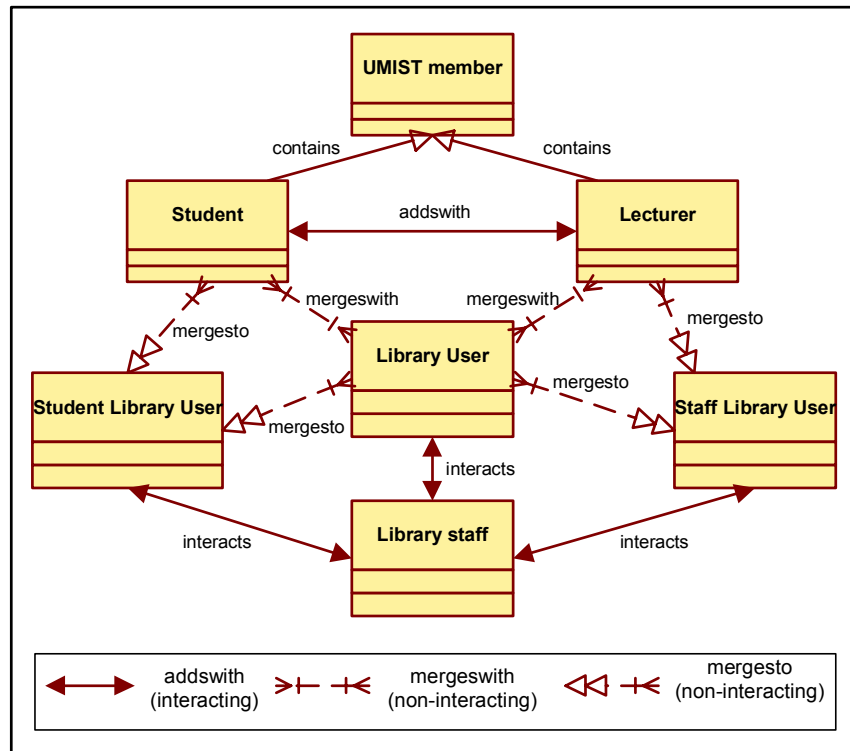


Figure 5.11: An example of collective behaviour reuse in RAMASD

To effectively use patterns in software design a pattern composition technique is required in addition to methods for selecting suitable patterns [214]. In this respect, RAMASD supports effective use of patterns since patterns are represented by role models that can be systematically composed based on the role algebra.

5.5.2.2 An Example of Behaviour Reuse

An example of how common behaviour can be reused in RAMASD is given in Figure 5.11. The example considers an ABS to support the operations of a university. Members of staff as well as students are also members of the university community and have access to the library. However, they have different access to library resources, for example, members of staff can borrow books for one year without any need for renewal whilst students borrow books for only one month.

In existing ABS design approaches which use roles, this would be difficult to represent and it could be generally done in two ways:

- An obvious way to do that would be to manually introduce separate roles for each possible library use, i.e. *StudentLibraryUser* and *StaffLibraryUser*. However, the basic collective behaviour corresponding to the interaction of a library user would be repeated twice. Furthermore, a new role would be required every time some modifications to the original role were required.

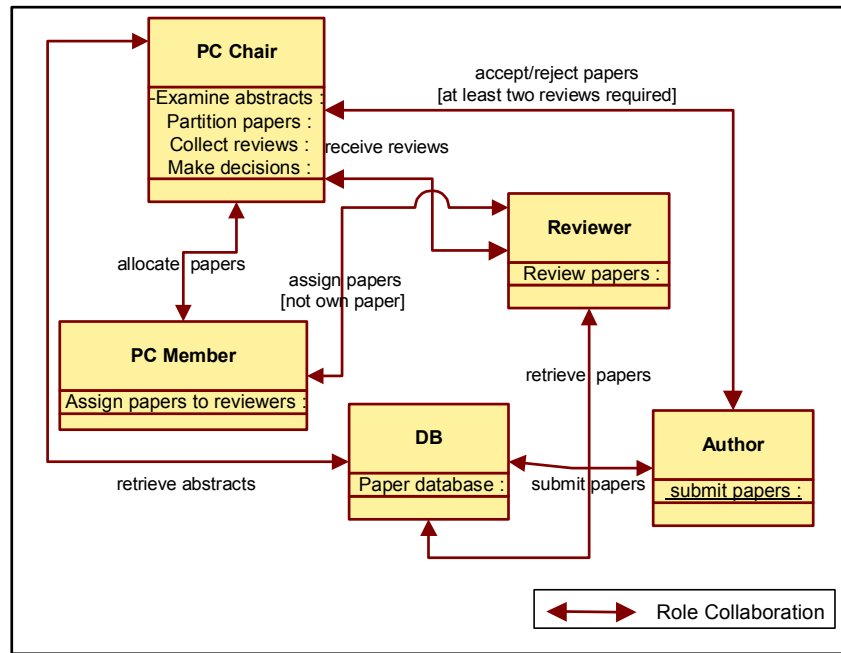


Figure 5.12: RAMASD roles for the conference management system example

- b) Another way would be to modify the role *LibraryStaff* so that it would provide different access privileges to *LibraryUsers* that are *Lecturers* to those which are *Students*. Such a solution would result in making different assumptions for the *LibraryStaff* role at a low level of detail and hence increasing the complexity of the specification. Furthermore, it completely lacks generality. For example, if the library roles were to be used to design ABS supporting a community library then different low level assumptions would need to be done again.

RAMASD offers an innovative way to handle such cases, which is introducing an appropriate merging relationships between the related roles. For each combination of roles that result in a different joint behaviour, a third role can be introduced. However, instead of the designer having to specify the differences in behaviour manually each time, this can be done automatically by the role allocation algorithm. For example, when an agent plays both *Lecturer* and *LibraryUser* then in effect it will be playing the *StaffLibraryUser* role. However, the designer does not need to explicitly include the *StaffLibraryUser* role in the design. If the roles and their merging relationships had been previously specified and stored in some storage area, the role allocation algorithm should be able to include *StaffLibraryUser* in the design automatically.

In a similar manner, more combinations of the relations included in RAMASD can be used to simplify modelling of complex behaviours. For example, frequently used roles can be specialised and roles can exclude or require other roles. It is expected that an ABS designer will

specify a number of relations among certain roles in an application domain and then reuse them as required.

5.5.3 Representing Organisational Settings

Representing organisational settings using roles has been widely used in the ABS systems research community. RAMASD represents organisational settings by appropriate organisational roles and by constraints based on inter-role relations and on role characteristics. This approach enables effective representation of both organisational patterns and organisational rules.

In a manner similar to other methods, organisational behaviour in RAMASD can be modelled by appropriate roles. The view is that assigning agents to play organisational roles will result to imposing organisational settings to the ABS. In order for organisational roles to be able to affect the agent behaviour, appropriate merging relationships between organisational and functional roles need to be specified. This is where RAMASD radically differs from existing approaches, for example [64].

To illustrate the above concepts, the conference management example discussed in [221] is used (Figure 5.12). The example involves designing an ABS to support management of conference paper reviewing processes. The conference system operates in the following phases: submission, review, decision, and final paper collection. During the submission phase, authors submit papers and are given a paper submission number. After the deadline for submissions has passed, the Conference Chair examines the abstracts of the submitted papers and delegates them to appropriate committee members for review. The committee members either contact referees and ask them to review a number of the papers, or they review them themselves if applicable. After the reviews are complete, recommendations are made to the conference chair who for each paper decides whether to accept or reject it. After the decisions are made, authors are notified of the decisions and are asked to produce a final version of their paper if it was accepted. Finally, all final copies are collected and printed in the conference proceedings.

The conference management system is an open system, and therefore there could be cases where agents could attempt to display opportunistic behaviours, that would benefit their owner to the detriment of the system as a whole. Such behaviours could include reviewing ones own paper or unfair allocation of work between reviewers. In [221] a number of example organisational rules are described aiming to prevent such situations. They include:

- Each reviewer must not review the same paper more than once.
- A paper author must not review his own paper.
- Two reviews are needed before a paper is accepted or rejected.

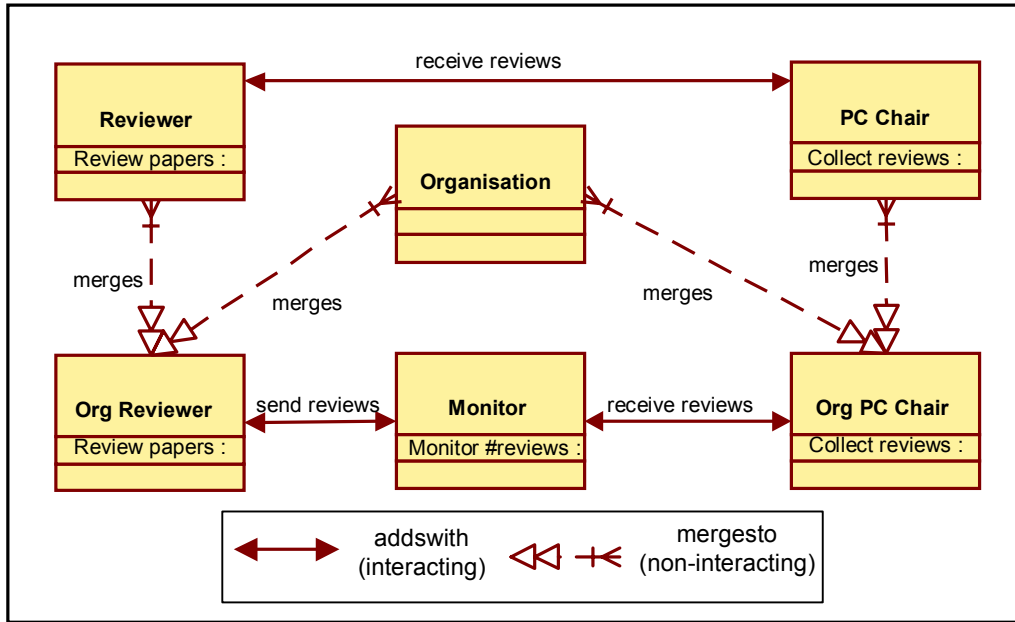


Figure 5.13: Enforcing organisational rules by appropriate merging of roles

Such rules are described in [221] using first order temporal logic. However, there is no actual description of how such rules would be mapped to analysis and design constructs. The traditional way of enforcing those rules would be to interweave the organizational rules into the individual roles. For example, the only place to check that at least two reviews were completed before the decision to accept or reject a paper was made is in the PC Chair role itself. This has two main weaknesses: Firstly, the role would require complex changes and it would not be directly reusable in other applications, and secondly this type of modelling would not assure that a newly arrived agent claiming that it plays the role would actually observe the organisational rules. For this reason it has been suggested that organisational rule enforcement should be done outside the ABS [150]. To the author's knowledge, the latest advance in this direction is by explicitly modelling organisational tasks and requiring application roles to execute them [48]. However, the linking of application roles to organisational tasks is done at a low level of abstraction involving increased complexity and there is no systematic technique to reuse such linkage.

In RAMASD, this issue is addressed by merging application roles with organisational roles. For example, in Figure 5.13 the role *Reviewer* is merged with the role *Organisation* resulting to the *Org Reviewer* role. The *Organisation* role represents behaviour conforming to particular organisational rules, which in this example refers to using the *Monitor* role as an intermediary for all interactions. Therefore, *Org Reviewer* has the same functionality as *Reviewer* with the difference that all interactions with the *PC Member* and *PC Chair* roles are carried out via the

Monitor role. In this way, the *Monitor* role monitors if the organisational rules are observed or not and can take appropriate actions if required. The resulting system is depicted in Figure 5.14.

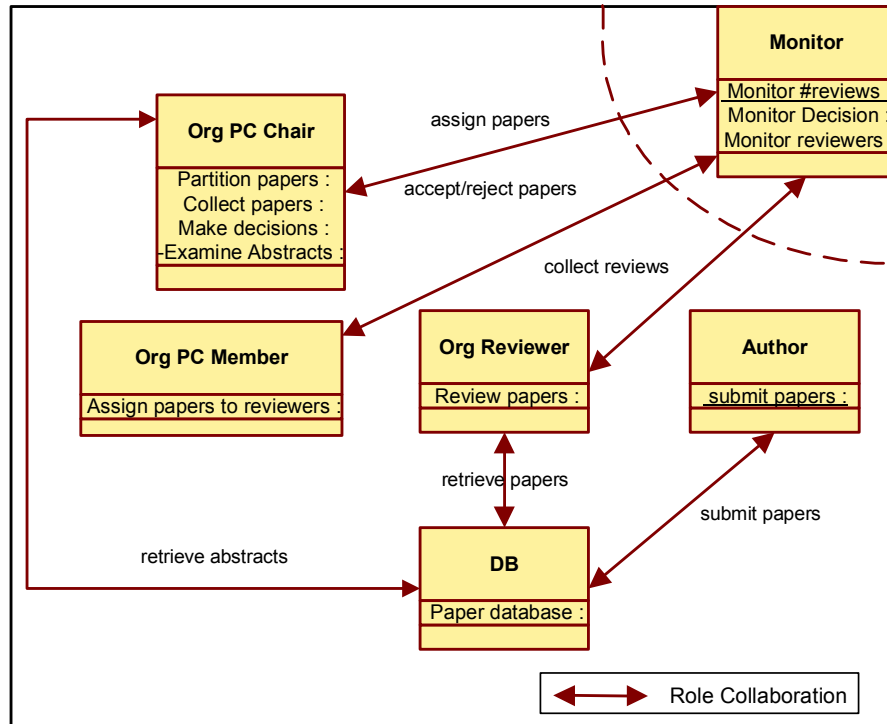


Figure 5.14: An example of modelling organisational rules using roles

5.5.4 Considering Non-Functional Aspects

Any software design effort should take into account non-functional aspects. In this work, non-functional aspects are represented in role models. This is done by introducing appropriate role models that contribute towards achieving certain non-functional qualities and by modelling non-functional requirements as quantitative constraints on certain agent characteristics. Agent characteristics are associated with the respective characteristics of the roles played by the agent.

As mentioned in Section 3.3.4, approaches to considering non-functional aspects in software design can be product oriented or process oriented, where the former focus on considering non-functional aspects throughout the design process and the latter on evaluating whether the resulting software product meets quantitative non-functional constraints.

Process-oriented approaches originated from the fact that non-functional aspects cannot always be described in a quantitative manner. For example, an ABS can be required to have “high security” and “good performance”. Furthermore, there is the issue of conflict in non-functional requirements. It can be inefficient to complete a design of software architecture only to find that non-functional constraints are not satisfied. Therefore, the process-oriented view aims at addressing such issues early in the design process.

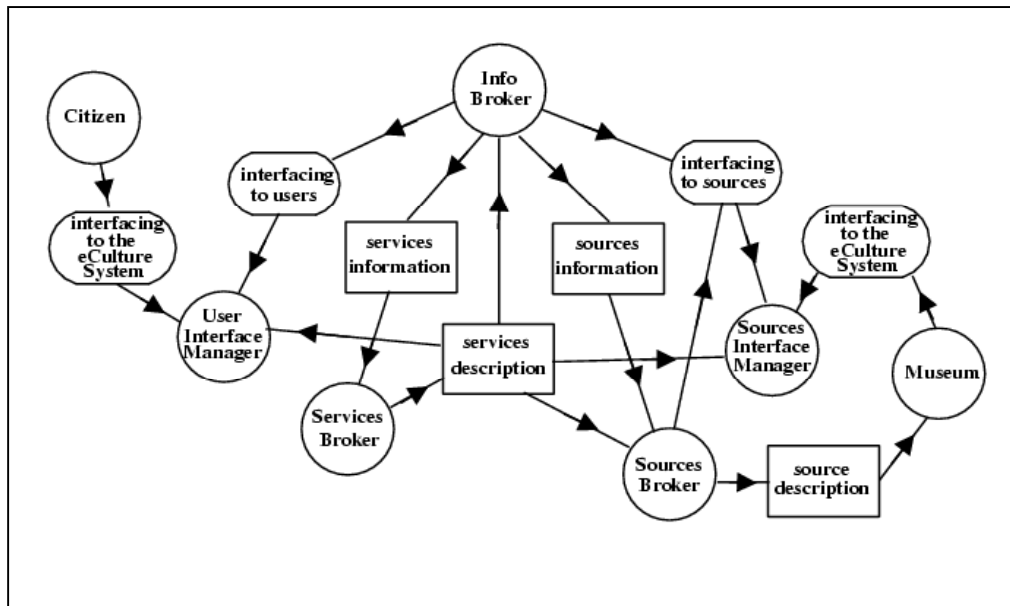


Figure 5.15: Extended actor diagram for an e-cultural system (aft Giorgini et al., 2001)

An example of ABS engineering methodology, where non-functional aspects are treated in a process-oriented manner, is Tropos [23]. In Tropos, appropriate *Actors*³, with special behaviour aiming to ensure fulfilment of non-functional requirements, are inserted in the conceptual models representing system activities (use cases) in the early design stages. Actors can contribute either positively or negatively with respect to the value of some non-functional aspect. The view in Tropos is that in this way, conflicts in contradicting non-functional aspects will be resolved as early as possible in the ABS engineering process. An example of a Tropos model where new actors are inserted is depicted in Figure 5.15, where actors are represented by circles. The example refers to the design of an ABS supporting provision of e-cultural services, for instance information about festival dates and theatre ticket booking. In the example given, the actors *Service Broker* and *Resource Broker* were inserted to increase the system extensibility. The new actors act as brokers for services and resources.

The problem with the approach followed in Tropos is that the solutions given cannot be directly reused in other applications. The inserted non-functional *Actors* are specific to each conceptual model. Furthermore, to the author's knowledge, there is currently no support from Tropos to automatically insert non-functional actors to the conceptual models. Hence, designers have to follow the design process step by step and introduce new actors as and when required.

³ Actors in Tropos are concepts quite similar to roles in the information systems engineering context described in Chapter 4.

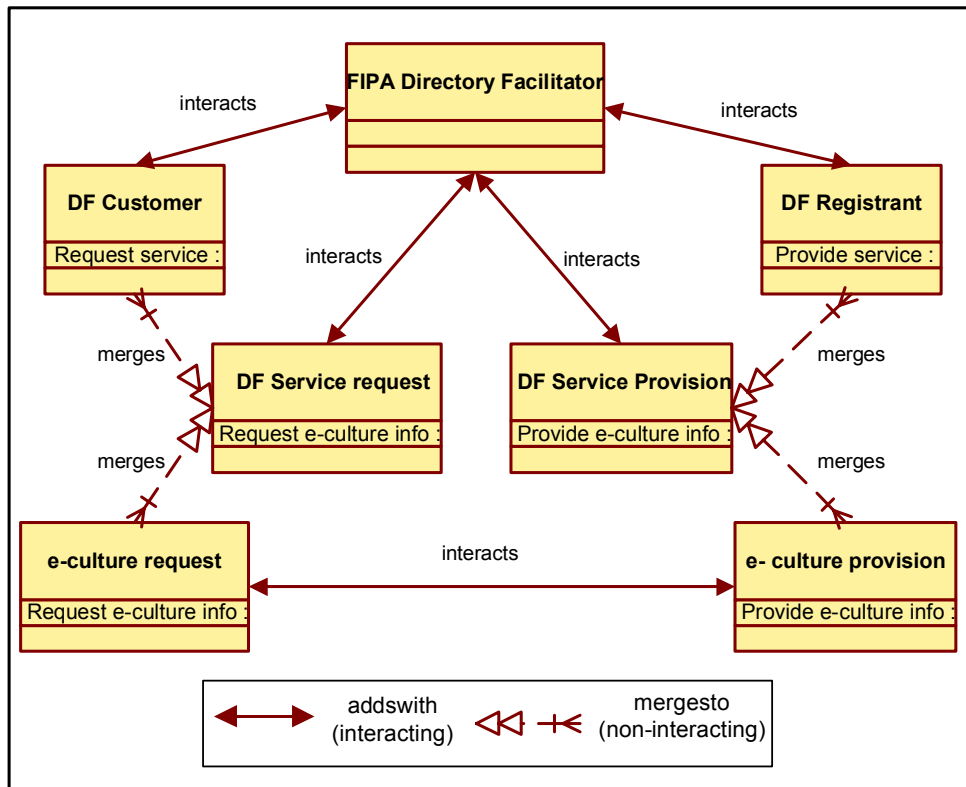


Figure 5.16: Using the FIPA directory facilitator role model for e-culture service brokering

The above weaknesses are addressed in RAMASD by considering non-functional roles. For example, in FIPA standards brokering is done by the Directory Facilitator [67]. This can be represented by a role model including the *Directory Facilitator*, *DF Customer* and *DF Registrant* roles. For an application domain, for example the e-culture domain, in order to be able to use the DF role model appropriate merging relationships should be specified. This is depicted in Figure 5.16 where the *e-culture request* merges with *DF Customer* and *e-culture provision* merges with *DF Registrant*. In this way, the designer needs to specify the above roles and their merging relationships only once and store them in a role model storage space. Then, whenever brokering is required, the designer would simply designate the use of the DF role model (in addition to the e-culture application model). Based on the merging relationships the appropriate *SF Service Request* and *SF Service Provision* roles would be also automatically retrieved from the role model storage space. Furthermore, allocation of roles to agents could be done automatically. In this way, both non-functional solutions are reused. This saves the designer from having to repeat design steps and to go into details, that is they are taken into account in an automatic manner leading to reduced design complexity.

Product oriented approaches involve evaluating design models, such as software architectures [114]. Generally, the idea is that for a non-functional parameter of interest, for example availability, to build models of software architectures that could possibly used to deliver the

system application functionality and reason about the suitability of these architectures for that particular performance parameter and application context. RAMASD supports a similar, yet relatively limited, view of quantitative modelling of non-functional aspects via the use of performance variables (see also Section 5.2.1.1). Performance variables can be used to represent various non-functional aspects including performance, availability and modifiability. The idea of performance variables is that certain non-functional aspects can be modelled as role characteristics. For example, in a PDA ABS the *User Interface*, *Personal Profiler* and *Document Handler* roles may be involved (Figure 5.17). *Personal Profiler* is associated with an amount of memory where the profile of the user is stored and document handler is associated with a database containing user documents. It is assumed the memory an agent will be associated with is equal to the sum of the amounts of memory associated with the roles the agent plays. As PDAs normally have limited amount of memory, in allocating roles to agent components it needs to be specified that the memory of each agent should be less than a particular threshold. This constraint would then drive allocation of roles to agent types.

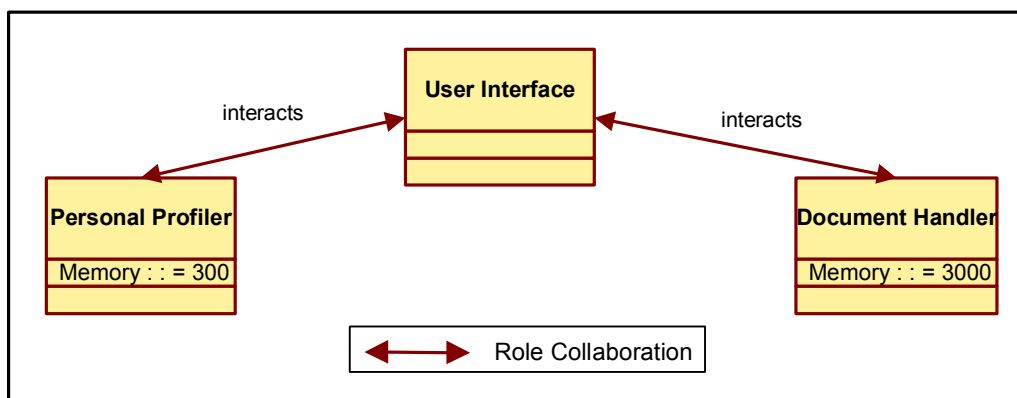


Figure 5.17: A personal assistant role model

5.5.5 Considering Design Heuristics

Design heuristics are instrumental in software design. In RAMASD design heuristics can be described either as constraints between roles or between agent and role characteristics.

In designing software architectures general design rules often need to be applied. Examples are low coupling and high cohesion [118]. Those heuristics assist in designing more understandable and efficient software architectures. Furthermore, several design heuristics concerning specifically the design of ABSs have been proposed. For example, in role-based design all roles requiring access to the same resource should be allocated to the same agent [38].

As mentioned previously, another heuristic proposed in [38] is *The Sphere of Responsibility* heuristic. According to this heuristic each agent should be responsible for controlling a number of resources or providing a number of services. This area of control is known as the agent's

Sphere of Responsibility. Therefore, when considering what agents will exist the developer will need to consider how the application domain will be partitioned. To illustrate this heuristic, in [38] they provide an example concerning a trading scenario involving four roles: *User Interface* for entering user preferences, *Negotiator* for carrying out the negotiation with traders, *Accountant* for settling payments and *Trade Consultant* for provision of trading expertise. In the examples given, two areas of responsibility were identified, one concerning the whole company and one concerning particular trading sectors. This is illustrated in Figure 5.18 where the presence of 4 trader agents, each with their own local sphere of responsibility, is assumed. Each trader agent will use the local computing resources to provide personalised user interaction and trading expertise. The negotiation and settlement activities have been centralised and placed in the company sphere, where they will be available to the whole company.

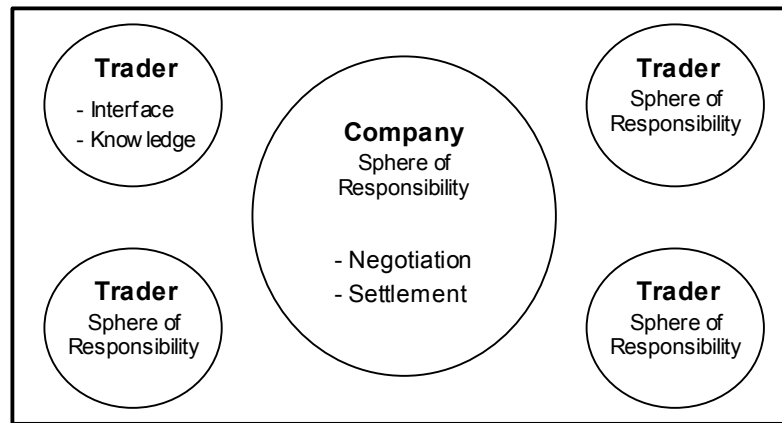


Figure 5.18: Spheres of responsibility (Collins et al. 1999)

In RAMASD such heuristics could be enforced by appropriate relations among roles. In the above example, the sphere of responsibility heuristic could be enforced by the introduction of two utility roles representing the two identified spheres of responsibility: *Company-wide* and *Trader-specific*. Those two roles exclude each other. Subsequently, any other role belonging to each of the two spheres of responsibility should require the appropriate utility role, for example, *User Interface* should require *Trader-specific*. In this way, a) an automatic allocation of roles to agents is possible by an appropriate algorithm, for instance the one described in Section 6.5, and b) the same role model can be reused to design other ABSs and the heuristic will be applied without any further effort from the designer. This example is depicted in Figure 5.19.

Similar results could be achieved by modelling each sphere of responsibility by an appropriate performance variable. For example, *SphereOfResponsibility*, which would be taking values of only 0 and 1. In this case, the heuristic could be enforced by requiring that the *SphereOfResponsibility* variable of every agent to be 0 or 1. This approach, however, would require to define how the values of the agent *SphereOfResponsibility* variables would be

obtained from the values of the respective variables of the roles played by the agents. This is not a trivial issue and, therefore, such specifications are outside the scope of RAMASD as discussed in Section 9.3.

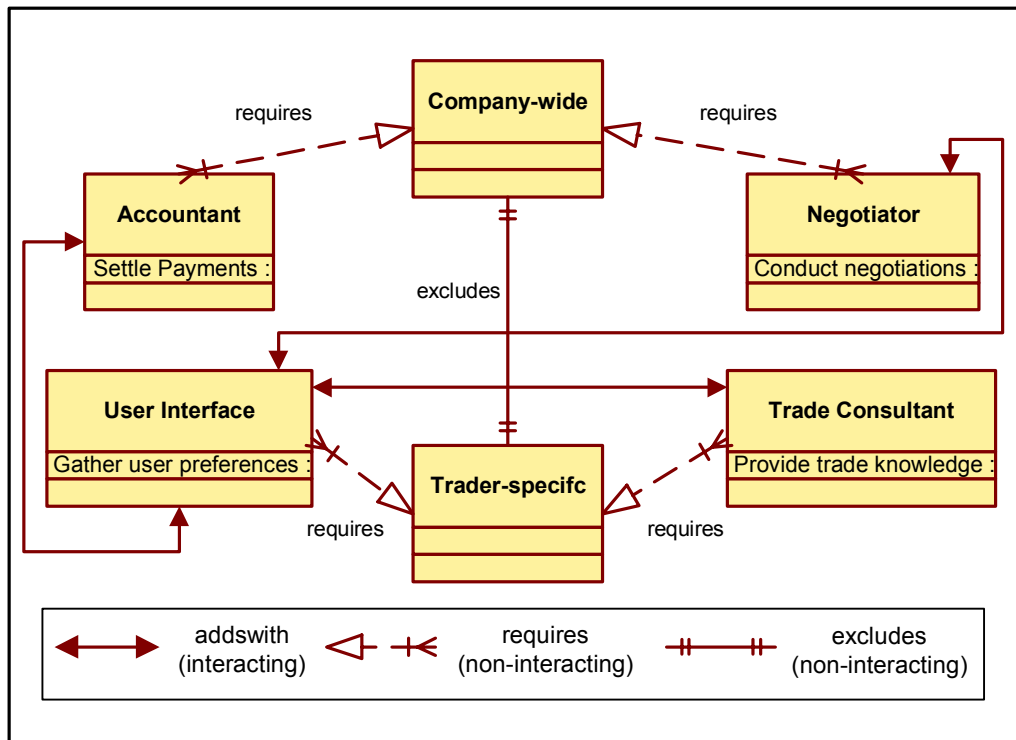


Figure 5.19: Specifying the spheres of responsibility heuristic using role relations

5.6 Using RAMASD with Existing Methodologies

An important advantage of the RAMASD method is that it can be used in conjunction with existing ABS engineering methodologies. The degree of compatibility between RAMASD and existing methodologies is high or low depending on whether they use the role concept for modelling the agent-behaviour.

High compatibility exists when the methodologies include the role concept as is the case with the majority of informal approaches, for example MESSAGE/UML, SODA, Gaia and Zeus. RAMASD can be used after the analysis stage to drive the design of the agent components. This requires formulating the role allocation problem in RAMASD, including the definition of constraints on roles and role characteristics and the possible introduction of additional role models to represent organisational and non-functional aspects. After role allocation, the remaining phases of each existing methodology can be followed. For example in SODA the topological model could be created and considered after designing the agent components using RAMASD.

The compatibility of RAMASD is low in methodologies where the role concept is not included, for example DESIRE. In such cases, RAMASD can still be useful but an additional role modelling phase would be required. For example, in DESIRE the tasks that would need to be carried out by the agents should be analysed according to some task analysis method, such as the one described in [32], and grouped to roles following general heuristics [109]. Subsequently, additional roles to represent organisational and non-functional aspects should be introduced as required and role allocation could take place. Finally, the tasks associated with each agent component could be determined from the roles allocated to it and the rest of the DESIRE methodology (task and component specification, code generation) could be applied normally.

5.7 Summary

Current methodologies for ABS engineering do not adequately support the transformation of analysis knowledge to design decisions that take place when designing ABSs, requiring the designers to manually address the complexity of the design problem. This makes designing large, real world, ABSs a tiresome and error-prone exercise.

To address these concerns, the RAMASD design method was introduced. RAMASD reduces design complexity by enabling designers to work at a high level of abstraction and by semi-automating the design process. RAMASD models agent behaviour using roles and it views ABS design as a problem of allocating roles to appropriate agents. Design requirements are represented by appropriate roles and design constraints on roles and role characteristics. Two innovative ideas behind RAMASD are to enable high-level design by defining the role concept so that it can represent a rich set of agent behavioural aspects and to use the synthesis concept to enable semi-automation of the design process. Those two ideas were fundamentally supplemented by the main innovation of RAMASD, the *role algebra*. The role algebra is a formal model of role relations concerning allocation of roles to agents. The semantics of this model are described using a two-sorted algebra. The role algebra supports both high-level design, enabling specification of design constraints at the role level, and semi-automation of the design process by enabling automatic role allocation after role selection has been made.

RAMASD enables reuse of organisational design knowledge by allowing designers to specify and retrieve relevant role models, and to manipulate them using the role algebra. Similarly, non-functional aspects and design heuristics can be described using roles and constraints on role characteristics and taken into account on role allocation.

Finally, RAMASD can be used in conjunction with a number of existing ABS engineering methodologies that make use of the role concept.

Chapter 6

Implementation of RAMASD

To test the applicability of RAMASD, it has been integrated into the Zeus agent building toolkit. A number of new toolkit components have been developed to implement the following extensions: a) a role model editor b) a specification language to specify design constraints and c) a set of algorithms to allocate roles to agents.

6.1 Extending Zeus to Support RAMASD

The Zeus agent-building toolkit was selected as the basis for the tool to support RAMASD since it already supported role modelling. It also provided a user friendly environment for ABS design and deployment and it had a modular architecture, which was easy to extend. To provide support for the RAMASD method, a number of new components interacting with the existing ones were developed. These components enable the designer to create, edit, store and retrieve role models, to specify appropriate design constraints and to automatically generate and deploy the ABS in the form of Java source code. The developed tool is referred to as the RAMASD-Zeus ABS design tool.

A high level overview of the RAMASD-Zeus is given in Figure 6.1. A main goal of this tool is to support ABS designers to represent previous design knowledge using role modelling and to place this knowledge in a repository, the role model library. The repository would then be used by subsequent ABS designers who wish to reuse subsystems or to modify and rebuild legacy systems using agent technology. To support role allocation steps in RAMASD, the tool uses appropriate algorithms to search for feasible solutions satisfying the design constraints. Finally, the resulting designs can be used to generate template systems linked to libraries of domain-specific implementation code.

The RAMASD-Zeus ABS design tool consists of three components:

- *The main designer GUI.* This provides editors for manipulating role models and design constraints and interfacing the role model library for storing and retrieving role models. Specification of design constraints is done in a simple specification language, the Role Constraint Language (RCL). Users of the RAMASD-Zeus tool can either use a user-friendly Java-based front-end to RCL or they can write RCL statements directly in a constraint specification file.

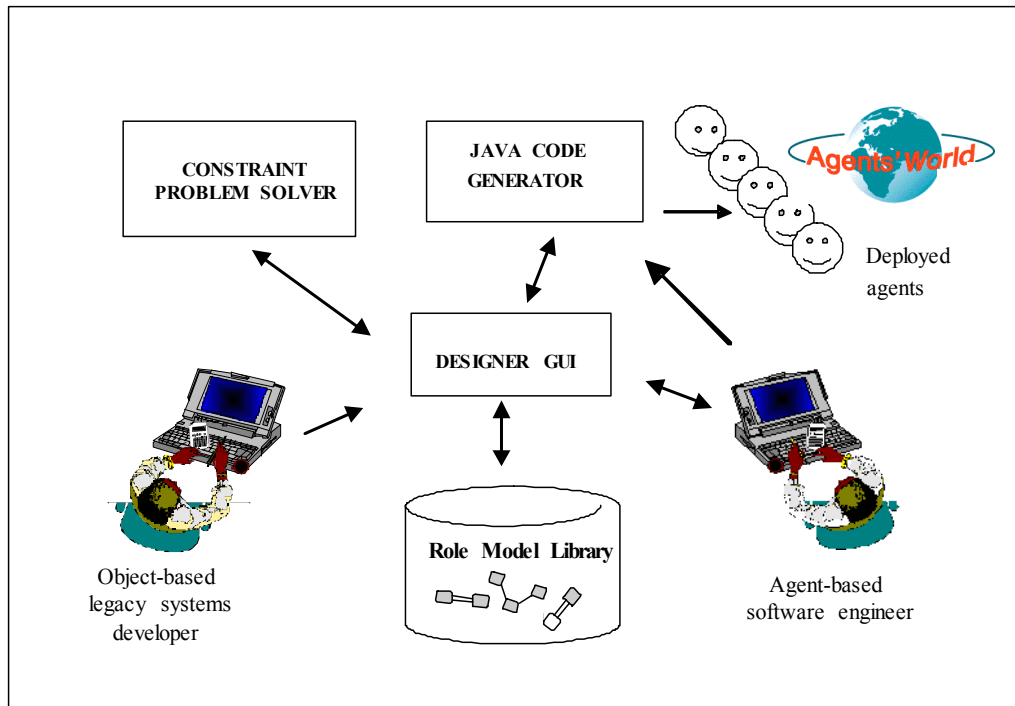


Figure 6.1: Conceptual view of the extended Zeus ABS design tool

- *The constraint problem solver.* This component implements a special purpose heuristic algorithm and various standard constraint problem solving algorithms for allocating roles to agents.
- *The Java agent generator.* This component provides algorithms for determining the agent characteristics based on the roles that have been allocated to the agents and for generating Java code for the deployment of the ABS.

As discussed in Chapter 5, the design of an ABS using RAMASD generally involves a number of iterations. The process of using the extended Zeus toolkit in conjunction with RAMASD can be described as follows: Legacy systems developers can use the tool to describe well-known design solutions (patterns) in an application domain in the form of role models. Those models are stored in a role model library. ABS engineers retrieve a number of role patterns from the role model library and customise them to suit the particular application being developed. Furthermore, they may introduce new role models as appropriate and store them in the role model library for future use. The next step is to specify any design constraints concerning desirable functional, non-functional and organisational aspects of the designed ABS. This is done using the role constraints editor, which is part of the extensions added to the Zeus agent building tool. Alternatively, constraints can be specified directly in a constraint specification text file. Subsequently, the constraint problem solver component searches for feasible allocations of roles to agents and notifies the engineers accordingly. If a solution is found, and it is considered appropriate, then the engineers launch the deployment step which involves

generating Java code from the Java code generation component. If a solution is not possible with the current specification, the designer can modify some of the problem parameters and request a new search to be made. This can be repeated until a satisfactory allocation of roles to agent types is done.

The remainder of this chapter is organised as follows: Section 6.2 summarises the Zeus agent building toolkit main components. In Section 6.3, the extensions done to the AgentGenerator component are described and the way that RAMASD can be applied using the extended AgentGenerator is illustrated. The specification of design constraints is done using a simple specification language which is described in Section 6.4. The algorithm used for allocation of roles to agents is described in Section 6.5 and the chapter concludes in Section 6.6.

6.2 The Zeus Agent Building Toolkit

The innate difficulty of constructing multi-agent systems has motivated agent developers to move away from developing point solutions to point problems in favour of developing methodologies and toolkits for building distributed multi-agent systems. This philosophy led to the development of the ZEUS Agent Building Toolkit [147] (downloadable from <http://more.btexact.com/projects/agents/zeus/>), which facilitates the rapid development of collaborative agent applications through the provision of a library of agent-level components and an environment to support the agent building process.

The ZEUS toolkit provides a set of components, written in the Java programming language, that can be categorised into three functional groups: an agent component library, an agent building tool and a suite of utility agents. Details of the Zeus Toolkit components are found in Appendix B.1.

6.3 Extending Zeus to Support RAMASD

To provide software support for RAMASD, it was decided to extend the Zeus ABS development methodology and software toolkit. RAMASD was integrated with the existing Zeus methodology amending the design and realisation stages. To provide appropriate software support, the existing Visual Agent Creator and the Code Generator components were extended and new components were developed.

6.3.1 The Extended Zeus Agent Development Methodology

The primary extension to the Zeus ABS development methodology — described in Appendix A.6 — proposed in this thesis is to use RAMASD to semi-automate the design process and to enable designers to work at a high level of abstraction. This spanned the design and the realisation stages of the initial methodology.

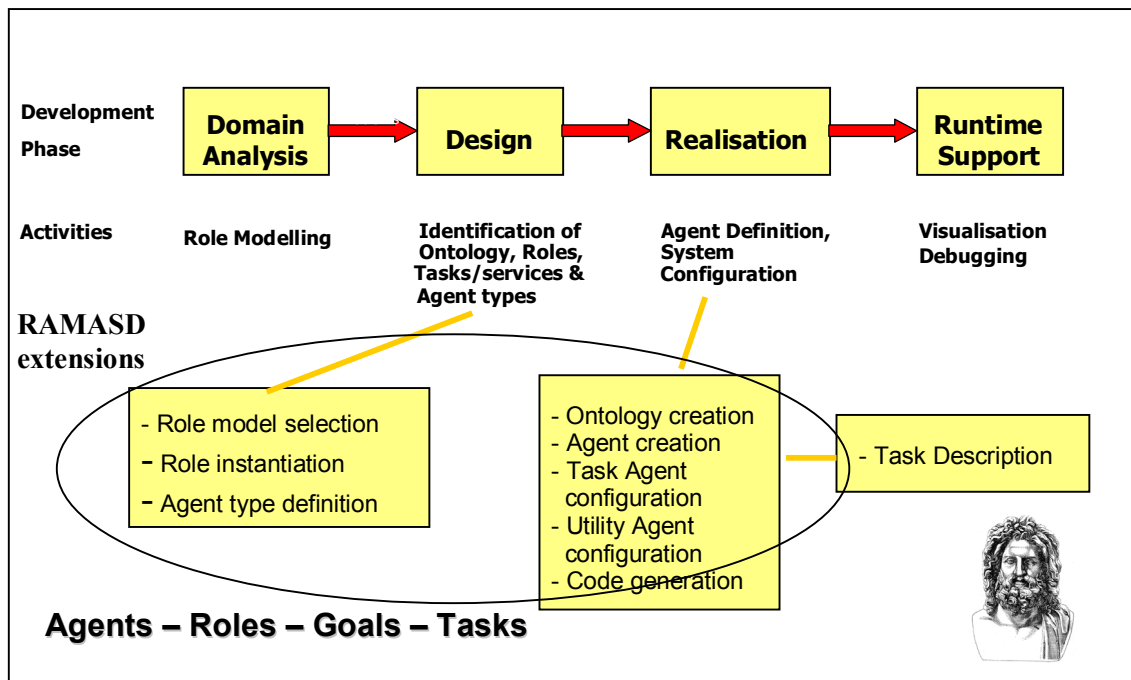


Figure 6.2: The extended Zeus agent development methodology

As can be seen in Figure 6.2, the design phase involves selecting appropriate role models, customising them for the application on hand and allocating them to agents. In the extended Zeus agent system development process, these steps are all done according to the RAMASD method, that is role allocation is done automatically by the Zeus tool.

The realisation phase is modified only as far as it concerns the agent creation stage. Agent definition is done automatically based on the roles each agent plays. This is because the agent tasks, resources and goals are defined by the respective tasks, resources and goals of the roles the agent plays. This is the case for the coordination protocols, the negotiation strategies and the organisational relationships of the agent as well.

In particular, the following steps need to be taken during the extended Zeus design phase:

- *Role model specification.* The role models that will be used are specified. This involves instantiation of reusable role interaction patterns and definition of role models specific to the application under development.
- *Role configuration.* The characteristics of each role, for example the resources it requires and the tasks it is able to perform, are specified. At this stage any performance parameters are also defined.
- *Task definition.* Tasks are defined in detail. This is done in a similar manner to that in the original Zeus agent design phase.

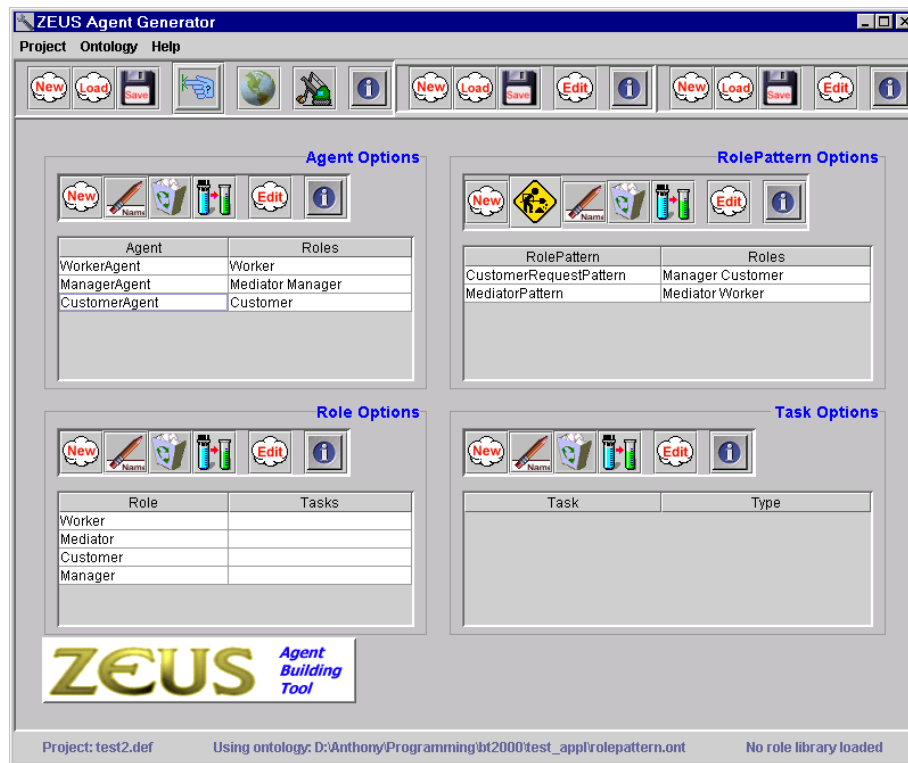


Figure 6.3: The extended Zeus Agent Generator component

- *Role collaborators:* The collaborators of each role are specified.
- *Role behavioural protocols:* The protocols used by a role to interact with other roles are specified.
- *Role compositional constraints:* The constraints that must be observed when a role is composed with other roles are specified. At this stage the performance parameters are assigned some value.

6.3.2 The Extended Zeus Visual Agent Creator Component

To provide support for the extended agent development process the *Zeus Agent Generator* and *Code Generator* components were extended. This required creating the *Role Constraint Editor* and *Role Allocation* sub-components and modifying the functionality of the existing ones where necessary.

The Zeus Agent Generator is the main component of the Visual Agent Creator tool. It provides links to all other Zeus components. For each application, the designer specifies a project including information about agents, tasks and Java code generation parameters. Project

definitions can be automatically translated to the *Zeus Frame based Language*⁴ and saved on disk. The Zeus Agent Generator component and the Zeus Frame based Language were extended to provide support for roles and role patterns (Figure 6.3).

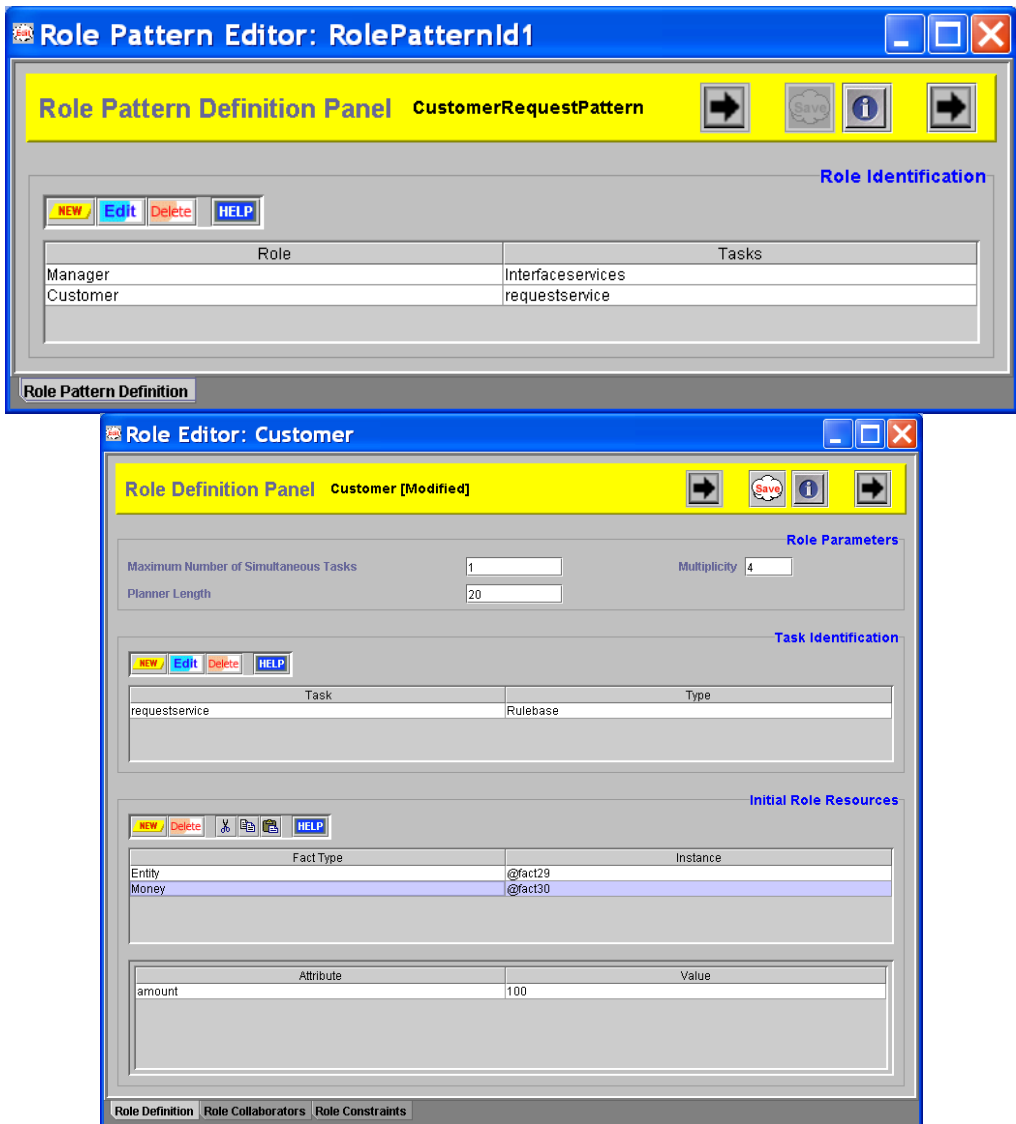


Figure 6.4: The role model and role definition editors

The Zeus Agent Generator component was extended to provide a suite of integrated sub-components that support specification of roles and role models in the extended Zeus ABS

⁴ The Zeus Frame-based language is a simple specification language in many aspects similar to XML. It has been introduced in Zeus before the release of XML. It was decided to use it in this project because its use is tightly linked with most parts of the Zeus Java class hierarchy and replacing it would require substantial effort which was outside the scope of a PhD project. There is currently ongoing work aiming to streamline Zeus knowledge representation mechanisms, which will replace this language with XML

design approach and allocation of roles to agent types. To facilitate ease of use, the editors have been designed to enable users to interactively edit roles agents by visually specifying their attributes. The current suite of editors includes:

- *Role Model and Role Definition Editors:* This is where the designer specifies role models and role characteristics (Figure 6.4). The role model editor allows for specifying role model characteristics. This involves specifying the name of the role model and the roles it comprises. Furthermore, the role model editor provides an interface to the *role model library*. The role model library is a component where role interaction patterns can be edited, automatically translated to some extension of the Zeus frame-based language and stored on disk. The role model library component aims at providing assistance in reusing design knowledge.

The role editor allows for specifying the tasks, initial resources (facts), and the planning length of each role. Furthermore, within role editor the collaborators of each role can be specified. Finally, role algebraic constraints can be introduced from this point by invoking the Role Constraints Editor described below.

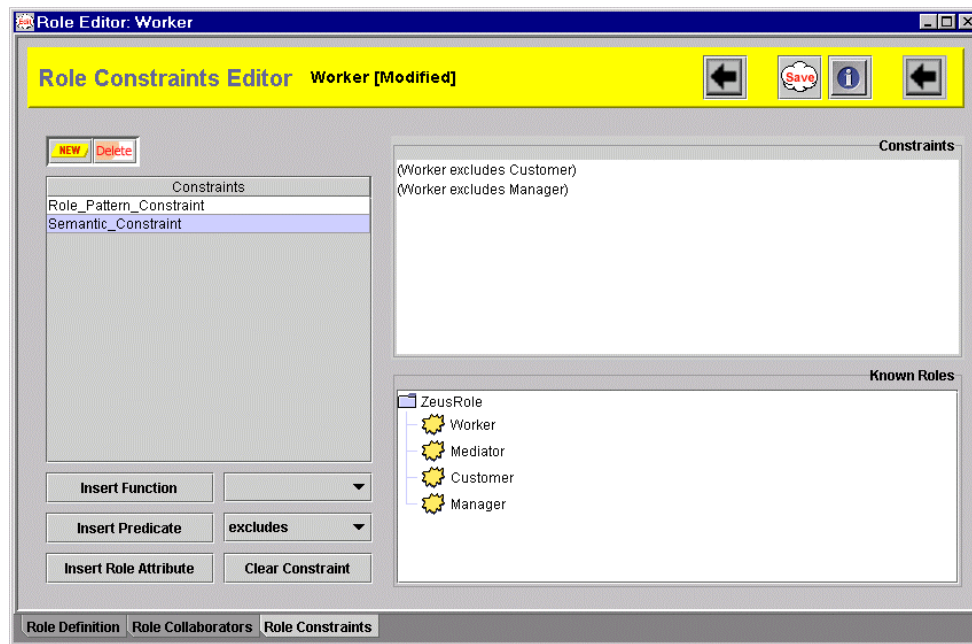


Figure 6.5: The Role Constraints Editor component

- A *Role Constraints Editor* (Figure 6.5) provides a graphical interface for specifying inter-role constraints. The designer is able to select role relations and available role names from appropriate pop-up menus. Constraint specifications can be stored and retrieved from constraint files (in the Zeus frame-based language format. Constraints are described in *RCL*, a simple constraint language which is described in more detail in Section 6.4. *RCL* is based

on the role algebra introduced. *RCL* provides a convenient user interface where designers can edit and manipulate various types of constraints in *RCL*. When an RCL specification is retrieved, a check for consistency is done using an RCL interpreter component.

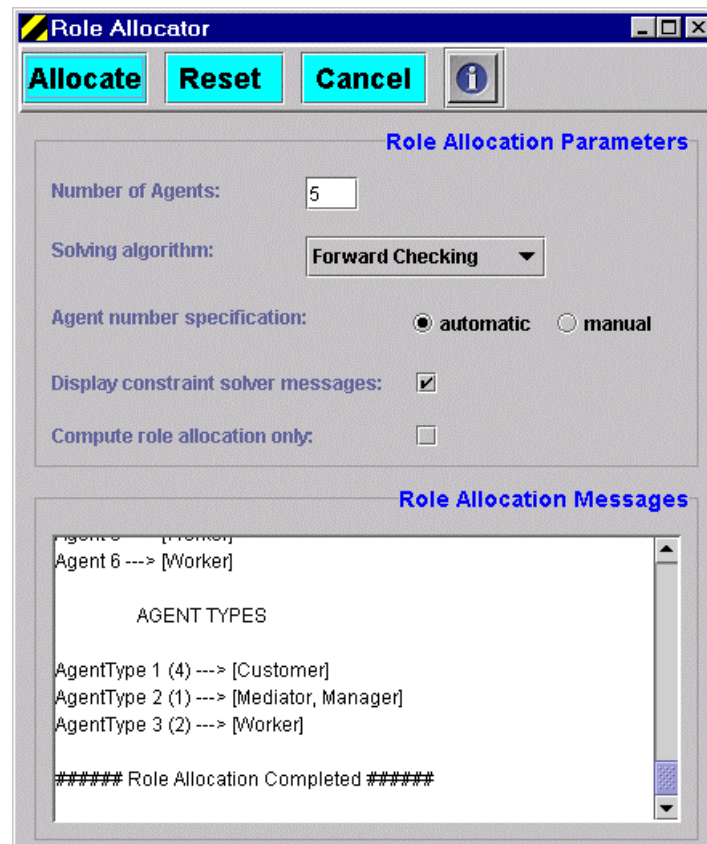


Figure 6.6: The Role Allocation component

- A *Role Allocation Component* (Figure 6.6) allows the designer to experiment with different role models and design constraints until a satisfactory design has been reached. The role allocation component formulates and solves a constraint satisfaction problem based on compositional constraints. Currently, the search algorithm described in Section 6.5 and three known search algorithms for constraint satisfaction problem solving have been implemented: *simple backtracking*, *simple backmarking* and *forward checking*. Those algorithms are executed with interfacing the Java Constraint Library [14]. In the current version of the role allocator component, the designer is able to specify the number of agents that need to be produced, the search algorithm that will be applied, to see detailed messages of the search process and automatically proceed to generating Java code for the designed agents.

In the case that no feasible solution is found, then the designer can try a different search algorithm or they can go back to the role model definition and role constraint editors and

change details on role characteristics and the various role constraints, in line with the synthesis-based semi-automatic process discussed in Section 5.4.

The above extensions were implemented on the Zeus agent building toolkit version 1.2.0 using JDK1.3. The JavaCC parser generator [95] was used for the extensions to the Zeus frame-based language and *RCL*. Furthermore, the Java constraint Library [14]. was used for the three search algorithms as mentioned above.

6.4 The RCL Constraint Language

After identifying roles in the application domain and modelling non-functional aspects using well-known role interaction patterns, the next step is to model the remaining functional and non-functional aspects using constraints on roles and agent and role characteristics. This can be done using Role Constraint Language (RCL). RCL is a simple declarative specification language that was introduced to represent design constraints on roles and agent and role characteristics. The RCL syntax is simple and intuitive in order to facilitate the specification of constraints among roles. The underlying RCL semantic model is based on the algebraic semantics of the role algebra presented in Section 5.2.7. The basic concepts of RCL will be illustrated using the specification of the design constraints presented in Figure 6.7.

An RCL specification contains sections corresponding to role definitions, role constraints and general constraints. In the role definitions section, the names of the roles that will be considered in the multi-agent system design are specified. More than one role name can be specified in the same specification statement. Furthermore, it is possible to associate role characteristics, for example role collaborators and performance variables, with role names. This is illustrated in Figure 6.7 where it is specified by specifying that the role *Employee* has associated the integer performance variable *database* and the string performance variable *negotiation*. At the same point, any role characteristics are assigned values. The syntax for referring to the characteristics of each role is similar to that of common programming languages, using a ‘.’ to link the role name and the role characteristic name.

In the role constraints section any constraints between roles are specified in prefix form. For example, the fact that the *Manager* role contains the *Employee* role is denoted by `in(manager, employee)`. In this way, using appropriate constraints between roles the way that different roles should be allocated to agent types can be specified. For example, the *Excludes* relation is used to specify that an agent that is *Customer* cannot also be *Employee*.

In the general constraints section generic constraints concerning role characteristics are described. For example, in Figure 6.7 it is specified that all agents should have a *database* less than or equal to 15. In this project, it has been assumed that all agents have the same

characteristics as all roles that are assigned to them, and that the respective values are given by simple models. For example, in Figure 6.7 it is assumed that the value of the agent performance variables equals to the sum of the values of the respective values of the roles allocated to the agent. Hence, restricting the values of the agent performance variables affects allocation of roles to agents.

```

/* ROLE DEFINITIONS */

/* defining the roles involved in the application domain */

Role customer, manager;
Role employee {
    int database;
    string negotiation = "exponential decay";
}

employee.database = 10;

/* ROLE CONSTRAINTS */

/* specifying any inter-role constraints */

not(customer, employee);
in(manager, employee);

/* GENERAL CONSTRAINTS */

/* specifying any constraints on role characteristics */

Constraint Y {
    forall a:Agent {
        a.database <= 15
    }
}

```

Figure 6.7: Parts of an RCL specification

The EBNF syntax of the RCL language is described in Appendix C.

6.5 Allocating Roles to Agents

The search for a feasible allocation of roles to agents can be done using various algorithms. In this section, a simple custom algorithm that can be used to allocate roles to agents is described.

The algorithm aims to minimise the number of agent types produced. Therefore, it tries to allocate as many roles to an agent type as possible before moving to the next one. Merging roles are processed first and the algorithm moves to the remaining roles only when all roles that are

involved in a *Mergeswith* constraint have been allocated to an agent type. All roles are allocated to an agent type at least once. An overview of the algorithm is given in Figure 6.8.

1. Create a new agent type t .
2. While there are remaining unprocessed merging relationships:
 - a. Create agent type $t' = t$.
 - b. Allocate roles r_i involved in a merging relationship m to agent type t' .
 - c. If t' is consistent for some allocation combination of r_i to t' :
 - i. Check any constraints on the performance variables of agent type t' .
 - ii. If they are satisfied:
 1. $t = t'$.
 2. Goto step 2.
 - d. If agent type t is empty then error.
3. While there are remaining unallocated roles:
 - a. Create agent type $t' = t$.
 - b. Allocate a role r to the agent type t' .

Figure 6.8: A simple search algorithm for allocating roles to agent types

The algorithm starts with an empty agent type and randomly allocates it with a group of merging roles which are roles involved in a *Mergeswith* relation. Then, it checks whether those merging roles are involved in any other role constraints. If they are, then the additional roles involved in those role constraints are also allocated to the agent type. Subsequently, the algorithm checks whether the agent type is consistent, which involves examining whether all constraints concerning the roles so far allocated to the agent type are satisfied.

If those role constraints are satisfied, the next step is to check whether any general constraints concerning role characteristics of this agent type are also satisfied. If this check is successful, the roles involved are considered part of this agent type and the two steps above are repeated for as long as there are still merging roles to allocate. If allocating any further merging roles to the agent type results in constraints that are not satisfied, then a new agent type is created and the process is repeated. If there are constraints that are not satisfied and the agent type was initially empty, the algorithm stops with an error message.

When all merging roles are finished and there are still roles available, then the algorithm attempts to allocate the remaining roles to the current agent type. In case of failure, a new agent type is created and the process continues until all available roles have been allocated to agent types.

This is a simple and intuitive algorithm, which is currently used as a base line for comparisons in our ongoing research about role allocation algorithms. The algorithm has been shown to work reasonably well for case study examples involving approximately 40 roles and having, on average, 10 merging role constraints, 20 other role constraints and 2 general constraints. However, the algorithm becomes inefficient when the total number of roles increases, the number of merging role constraints decreases or the total number of constraints increases. As described in Section 9.4, detailed exploration of possible search algorithms is outside the scope of this thesis but could be the subject of future work.

6.6 Summary – Conclusions

In this chapter the implementation details of integrating the RAMASD approach in the Zeus agent building tool were described. The main components developed were the role model editor and the role allocator component. These were accompanied by RCL, a specification language to specify design constraints, and a set of algorithms to allocate roles to agents. The extended Zeus agent building tool was used to test the applicability of the RAMASD method in the case studies described in Chapter 7.

Chapter 7

Case Studies: Mobile Workforce Support and COVISINT

This chapter demonstrates the application of RAMASD to the design of ABSs for two case studies. It presents the role models and the design constraints elicited from the respective application domains and it gives an overview of the results obtained. The results demonstrate that it is feasible to apply RAMASD in real-world applications.

7.1 Applying RAMASD to Real World Cases

In the previous chapters, the use of formal algebraic relations among roles as a discipline for driving allocation of roles to agents has been discussed. The discussion was followed by a presentation of RAMASD, a semi-automatic approach for systematically selecting roles and allocating them to agent types. In this chapter, the RAMASD approach is applied to two case studies and the empirical results and observations are presented.

The purpose of this exercise is twofold. Firstly, it is to demonstrate that it is plausible to apply the RAMASD method to non-trivial real-world applications. The second is to provide evidence that will enable a satisfactory evaluation of the utility of the RAMASD method, which is done in Chapter 7.

The first case study concerns support of BT's mobile workforce. BT has about 25,000 mobile workers performing about 150,000 repair tasks everyday across the UK. BT is very much interested in appropriate technology enabling mobile workers to deliver high productivity and quality of service while lowering the operational costs. Support for BT's mobile workforce has many dimensions including travel management, teamwork coordination and work knowledge management. These three dimensions were considered in the first case study. This case study demonstrates how RAMASD handles quantitative non-functional aspects and design heuristics.

The second case study concerns COVISINT, a B2B electronic marketplace (B2B Exchange) concerning the automotive industry. COVISINT offers support for supply chain management, collaboration among automotive market business parties, procurement, quality control and corporate financial processes. This case study was selected to demonstrate how RAMASD handles qualitative non-functional aspects and organisational settings.

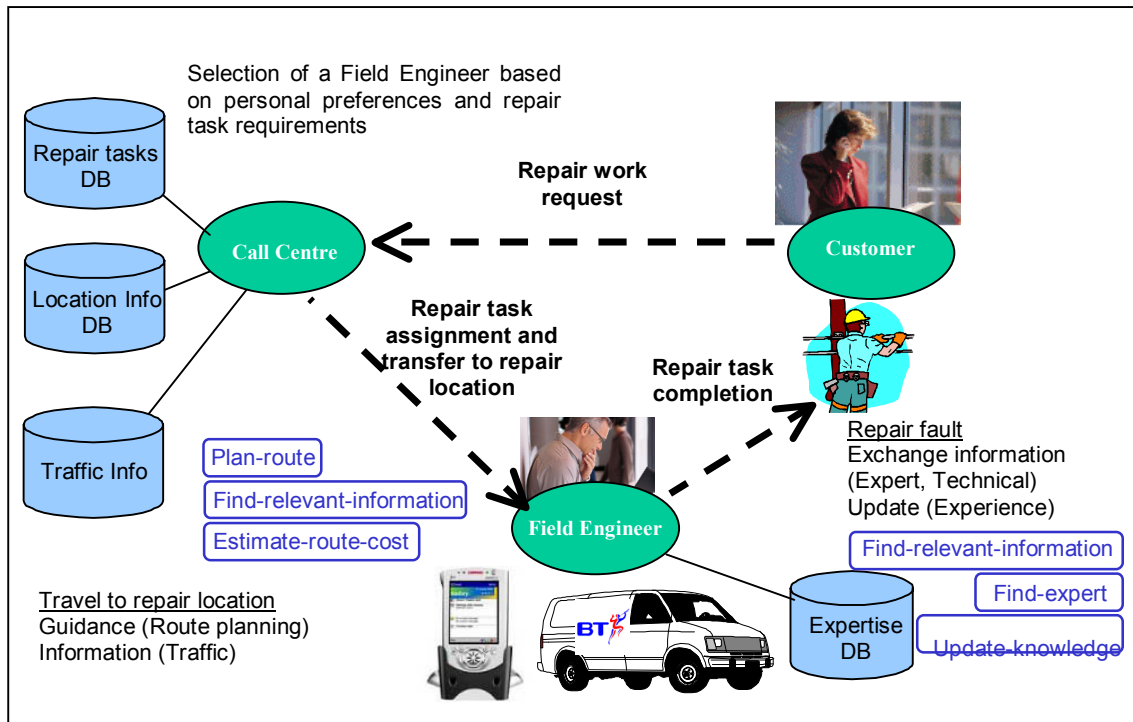


Figure 7.1: A high level view of the mobile workforce coordination case study

The remainder of the chapter is organised as follows: Section 7.2 provides an overview of the mobile workforce coordination application and it discusses the results of applying RAMASD in that application context. Similarly, Section 7.3 describes the application of RAMASD to the COVISINT case study. Finally, Section 7.4 concludes the chapter.

7.2 Mobile Workforce Support

Coordination of a mobile workforce is an issue of major concern in contemporary business organisations [28, 117, 175]. A typical example of a mobile workforce support problem is that of supporting repair engineers of telecommunication companies, such as British Telecommunications. In this section, the use of RAMASD is demonstrated based on a subset of the requirements for an ABS to support telecommunication field engineers.

7.2.1 The Mobile Workforce Support Problem

The aim is to build an agent system that would assist field engineers to carry out their work. Among the issues involved in such a system are those of *Travel Management*, *Teamwork Coordination*, and *Knowledge Management* [187, 199].

Provision of service to business and residential customers, network maintenance and fault repair are core activities of large telecommunications and IT companies. For example, British Telecommunications (BT) employs more than 20,000 field engineers across the UK to maintain networks, repair faults, and provide service to customers. Such companies are faced with a the

challenge of improving customer service across a diverse and broadening product range while achieving improved productivity and attendant cost savings. This requires effective computer support for the distributed workforce.

The key issues that need to be addressed regarding the supporting of mobile workforce are allocating the appropriate repair tasks to engineers at the appropriate time, assisting them in travelling to the fault location, enabling engineers to reuse past knowledge obtained through experience and storing new knowledge for later use. Travel management is about support to mobile workers moving from one repair task location to another. It involves finding the position of each worker, obtaining relevant travel information, planning the route to the next repair task location and allocating travel resources as required. Teamwork coordination is about allocating and coordinating the execution of repair tasks in a decentralised manner, taking into account the personal preferences and working practices of the mobile workers. Work knowledge management concerns storage and dissemination of expert work knowledge.

An overview of a repair scenario is depicted in Figure 7.1. The customer contacts the call centre and reports a fault in their telephone line and requests a repair. The repair request is logged and a search is done for a suitable field engineer to carry out the repair. This search takes into account the preferences of the engineer as well as their current location. Upon task assignment, detailed traffic information is obtained and a suitable travel route is recommended to the engineer aiming to minimise the transfer time. Subsequently, the engineer either completes the repair or consults a colleague or an appropriate knowledge base for assistance in difficult cases. Any new knowledge obtained through experience is added in this knowledge base.

7.2.2 Role Identification

In order to model the above scenario in terms of roles, the first thing to do is to start from use cases (see Section 5.2.1). For the purpose of the telephone repair request scenario, the following use cases are considered:

- *Teamwork coordination:* In this activity the customer places a request for a telephone repair. This request is placed in a pool of repair request tasks and it is eventually allocated to some mobile field engineer who will be responsible for its execution.
- *Travel management:* This involves providing up to date travel information to the field engineer including their current exact location, an optimal plan of the route to the next telephone repair task, as well as traffic information and managerial policy regarding travelling.
- *Work knowledge management:* Work knowledge management deals with maintaining and storing expertise for complex telephone repair tasks.

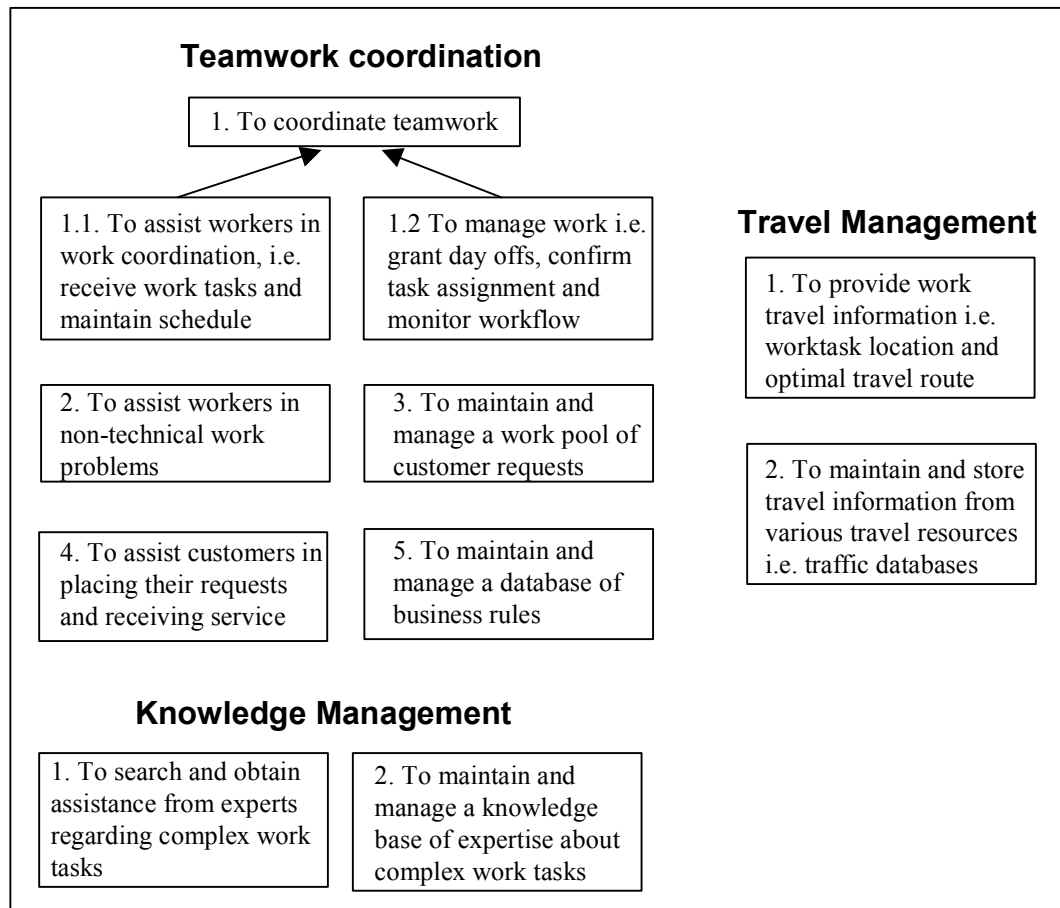


Figure 7.2: Use case goals for the telephone repair service teams case study

Each use case has a number of high-level goals depicted in Figure 7.2. The behaviour leading to achieving these goals can be modelled by appropriate roles. Hence, the following roles can be identified (Figure 7.3):

1. *Employee*: This role describes generic behaviour of the members of the customer service teams. An example of this type of behaviour is accessing common team resources including work practice announcements and business news.
2. *Coordinator*: The Coordinator role describes the behaviour required to coordinate the work of a field engineer. This includes bidding for and obtaining repair work tasks from a work pool, negotiating with other workers and the team manager as required and scheduling and rescheduling work task execution.
3. *Manager*: The Manager role models the behaviour of the team manager. This includes confirming task allocation, monitoring work and ensuring that business rules are followed.
4. *Mentor*: The mentor role provides assistance to field engineers for non-technical issues.

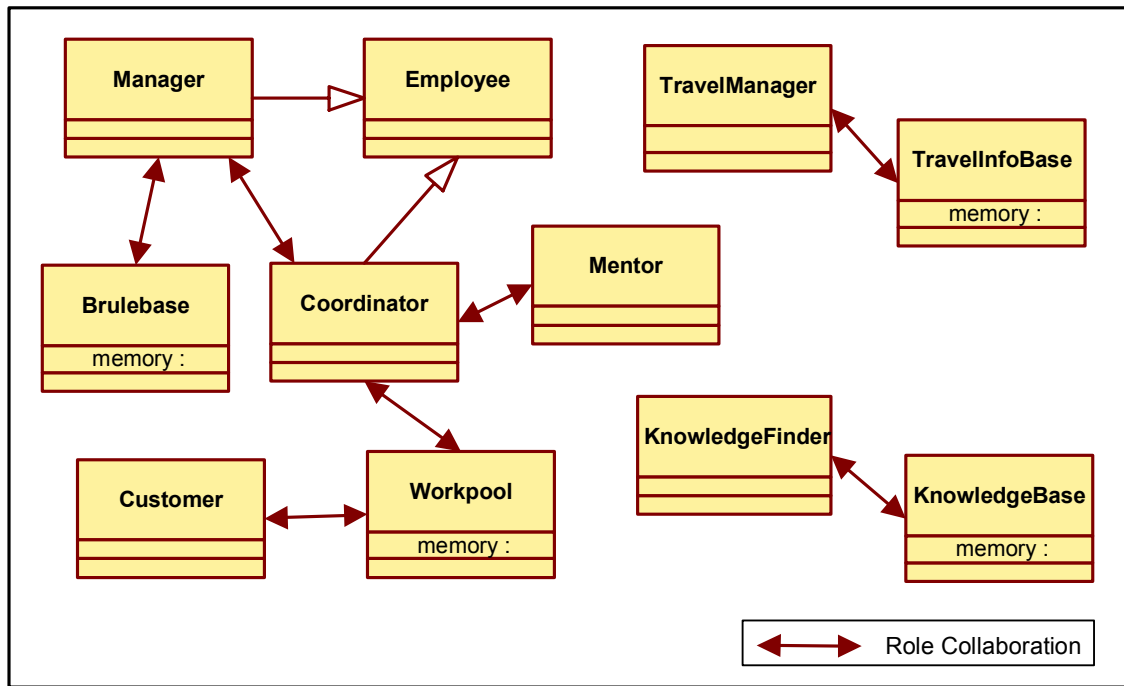


Figure 7.3: Role models for the telephone repair service teams case study

5. *WorkPool*: The *WorkPool* role maintains a pool of telephone repair requests. Customers interact with this role to place requests and engineers interact with this role to select tasks to undertake.
6. *Customer*: The *Customer* role models the behaviour of a customer. In involves placing telephone repair requests, receiving relevant information and arranging appointments with field engineers.
7. *Brulebase*: This role maintains a database of business rules. It interacts with *Manager* providing information about the current work policy of the business.
8. *TravelManager*: The *TravelManager* role provides travel information to the field engineer including current location, traffic information and optimal route to next telephone repair task.
9. *TravellInfoBase*: This role store travel information from various travel resources i.e. GPS and traffic databases.
10. *KnowledgeFinder*: This role searches for experts and obtains assistance regarding complex work tasks.
11. *KnowledgeBase*: The *KnowledgeBase* role maintains and manages a database of expertise about telephone repair tasks.

<code>/* ROLE DEFINITIONS */</code>	<code>/* ROLE CONSTRAINTS */</code>
<code>Ro employee, coordinator, mentor,</code>	<code>in(employee, coordinator);</code>
<code>customer, travelmanager,</code>	<code>in(employee, manager);</code>
<code>knowledgefinder;</code>	
	<code>not(customer, employee);</code>
<code>Role workpool, brulebase, workerassistant,</code>	<code>not(customer, travelinfobase);</code>
<code>travelinfobase, knowledgebase {</code>	<code>not(customer, knowledgebase);</code>
<code>int memory;</code>	<code>not(manager, coordinator);</code>
<code>}</code>	<code>and(mentor, employee);</code>
<code>workpool.memory = 1;</code>	<code>and(travelmanager,</code>
<code>brulebase.memory = 1;</code>	<code>knowledgefinder)</code>
<code>travelinfobase.memory = 2;</code>	
<code>knowledgebase.memory = 2;</code>	<code>merge(coordinator, travelmanager,</code>
<code>workerassistant.memory = 2;</code>	<code>knowledgefinder,</code>
	<code>workerassistant);</code>
<code>Role manager {</code>	
<code>collaborators = {Coordinator,</code>	<code>/* GENERAL CONSTRAINTS */</code>
<code>Brulebase};</code>	
<code>protocols = {contracting};</code>	<code>Constraint Y {</code>
<code>}</code>	<code>forall a:Agent {</code>
	<code>a.memory <= 2</code>
	<code>}</code>

Figure 7.4: Compositional constraints for the telephone repair service teams case study

7.2.3 Specifying Design Constraints

In Figure 7.4, compositional constraints for the roles described in Section 7.2.2 are specified in *RCL*. Apart from the application specific constraints, it includes constraints representing non-functional aspects and design heuristics.

In the telephone repair service teams example, roles that directly manipulate databases require access to some storage space. This is modelled by the performance variable *memory*. The memory requirements of each role are different. For example, *TravelInfoBase* and *KnowledgeBase* require twice as much memory as *WorkPool* and *Brulebase*. The memory requirement is an example of how non-functional aspects can be quantitatively modelled in RAMASD.

Part of the definition of the characteristics of the *Manager* role is shown in more detail in Figure 7.4. The collaborators of the *Manager* role are the *Coordinator* and *Brulebase* roles and its interaction protocol is the Contract Net. The *Employee* role is contained in both *Manager* and *Coordinator* roles. Furthermore, a *Manager* cannot coexist with *Mentor* or *Coordinator* and for security purposes a *Customer* cannot coexist with *Employee*, *TravelInfoBase* or *KnowledgeBase*. In order for an agent to be *Mentor* it must also be an *Employee*.

Furthermore, it is assumed that the point of interaction heuristic [38] needs to be applied in this example. According to this heuristic, common points of interaction should be allocated to the same agent. In the example considered, one such point of interaction is where the field engineer's personal assistant interacts with travel information sources to obtain Travel information and with the knowledge base to obtain expertise information about repair tasks. Therefore, it is required to have both these interactions carried out by the same agent. This can be specified in RAMASD as a requirement constraint between the roles *TravelManager* and *KnowledgeFinder* (Figure 7.4).

When an agent plays all three *Coordinator*, *TravelManager* and *KnowledgeFinder* roles, overheads in synchronising results from the three different activities — travel management, teamwork coordination and knowledge management — may occur. This is modelled as a merge of the *Coordinator*, *TravelManager* and *KnowledgeFinder* to the *WorkerAssistant* role. The *WorkerAssistant* role requires some memory to store intermediate synchronisation results — as specified in Figure 7.4.

7.2.4 Design Results

The application of the algorithm described in Section 6.5 to this design problem is straightforward. The algorithm starts from the sole `merge(coordinator, travelmanager, knowledgefinder, workerassistant)` constraint and creates an agent type owing those four roles.

Since `coordinator` contains `employee`, the `employee` role is also allocated to this agent type. The algorithm then checks the rest of the constraints included in this RCL specification and in this example they are all satisfied. Subsequently, the general constraint `a.memory <= 2` is also found to be satisfied since `a.memory = workerassistant.memory = 2`.

As there are no merging constraints left, the algorithm then attempts to allocate the remaining roles to the same agent type. Along this line, the `mentor` role is allocated to the current agent type as well. However, allocation of further roles is not possible since it would result in violation of certain constraints. More specifically, allocation of `customer` or `manager` is prohibited since they cannot coexist with `employee` and `coordinator` respectively.

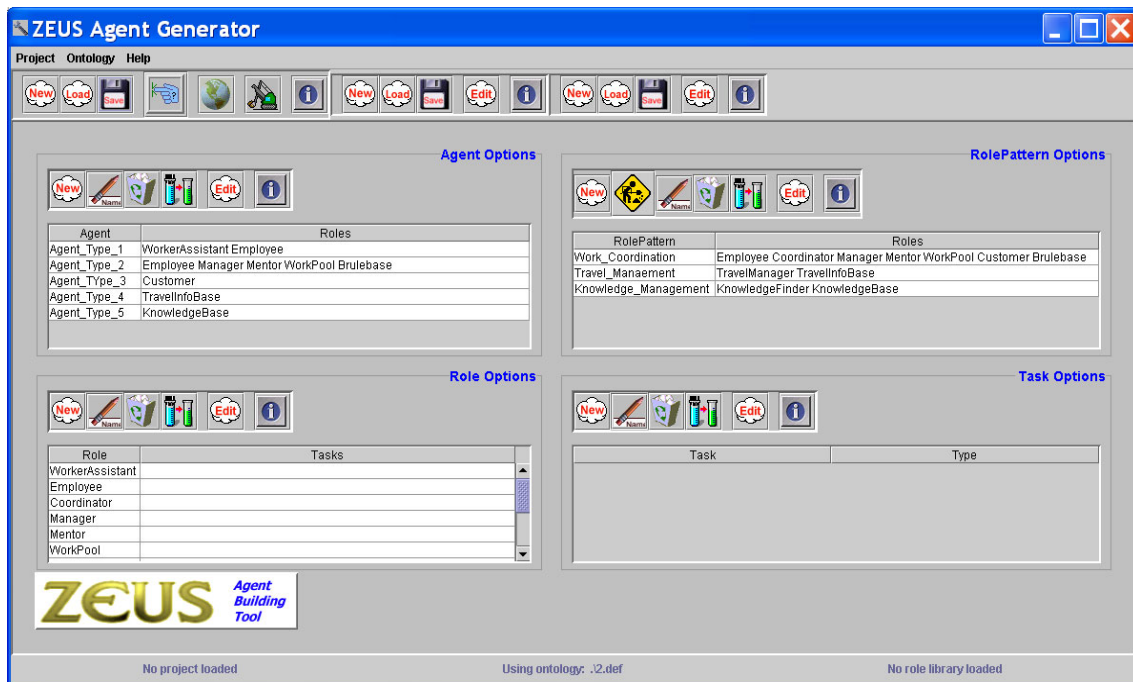


Figure 7.5: Snapshot of the extended Zeus toolkit for the mobile workforce case study

Furthermore, allocation of either `brulebase` or `workpool` roles would result to this agent type having a total amount of memory equal to 3, thus violating the memory constraint requiring that each agent type should have memory less than or equal to 2. The same reason does not allow allocation of `travelinfobase` and `knowledgebase` as this would result in the agent type having a memory of 4.

The algorithm then proceeds to create a second agent type and starts allocating the remaining roles to it. The first role to be placed in this new agent type is `manager`. Since `manager` requires `employee`, the `employee` role is allocated as well. Subsequently, the `brulebase` and `workpool` roles are allocated and all constraints are satisfied. These are all the roles that are allocated to this agent type as allocation of further roles is not possible. `customer` cannot coexist with `employee` and allocation of either `travelinfobase` or `knowledgebase` would result to the agent type having a memory of 4.

The algorithm then creates a new agent type and places `customer` in it. `customer` explicitly cannot coexist with neither `travelinfobase` or `knowledgebase` and hence, a new agent type is created initially containing the `travelinfobase` role. Allocation of `knowledgebase` to this agent type is prohibited since this would result in it having a memory of 4. Therefore, `knowledgebase` is placed on a fifth agent type and the algorithm ends.

A snapshot of the role allocation to agent types in the extended Zeus toolkit is shown in Figure 7.5

Agent Type 1		Agent Type 2		Agent Type 3	
has	plays	has	plays	has	plays
<i>Employee</i>	<i>Employee</i>	<i>Employee</i>	<i>Employee</i>	<i>Customer</i>	<i>Customer</i>
<i>Coordinator</i>	<i>Worker</i>	<i>Manager</i>	<i>Manager</i>		
<i>Travel</i>	<i>Assistant</i>	<i>Brulebase</i>	<i>Brulebase</i>		
<i>Manager</i>		<i>Workpool</i>	<i>Workpool</i>		
<i>Knowledge</i>		<i>Mentor</i>	<i>Mentor</i>		
<i>Finder</i>					
<i>Worker</i>					
<i>Assistant</i>					
<i>memory</i>	2	<i>memory</i>	2	<i>memory</i>	0

Agent Type 4		Agent Type 5	
has	plays	has	plays
<i>TravelInfo</i>	<i>TravelInfo</i>	<i>Knowledge</i>	<i>Knowledge</i>
<i>Base</i>	<i>Base</i>	<i>Base</i>	<i>Base</i>
<i>memory</i>	2	<i>memory</i>	2

Figure 7.6: Agent types for the telephone repair service teams case study

The resulting agent types for the mobile workforce case study are presented in Figure 7.6.

7.3 Example: An Automotive Industry B2B Exchange

To further illustrate the use of RAMASD, an example extracted from a large case study concerning an automotive industry B2B exchange is considered. The example is based on a simple B2B e-commerce model involving three business phases: quotation, negotiation and order fulfilment.

7.3.1 Case Study Overview

Automotive industry B2B exchanges are electronic business service providers offering a variety of services including business directories, auctions, supply-chain management and asset re-deployment and disposal [140]. The idea of such efforts is to bring companies from the automotive industry together and enable them to carry out their business in a more cost-effective and convenient manner using Internet technology. Automotive industry manufacturers are able to interact with their suppliers without having the burden of to interface different information technology systems. In addition to effectively transacting with their customers, suppliers are also able to consolidate their efforts, thus maximising their enterprise capability and the ability to pursue more business opportunities. In addition, all parties benefit from utility applications

available to B2B exchange participants, for example corporate services and customer relationship management software [57].

A representative example of an automotive industry B2B exchange is COVISINT (COllaboration, VIsion and INTegration) [42], which was initiated by DaimlerChrysler, Ford, and General Motors to create an optimised digital supply chain for the automotive industry. Additional drivers for the creation of COVISINT were the expected cost savings and the improved product life cycle management based on sophisticated software support. Currently, fourteen key players from the automotive industry have joined COVISINT together with two technology partners, Commerce-One and Oracle. COVISINT offers support for supply chain management, collaboration among automotive market business parties, procurement, quality control and corporate financial processes. The functionality offered by COVISINT is the basis for the case study considered in this thesis.

An important issue in business-to-business transactions is the underlying e-commerce model. For example, the standard Consumer Buying Behaviour (CBB) model [82], includes six stages: Need Identification, Product Brokering, Merchant Brokering, Negotiation, Purchase and Delivery, and Product Service and Evaluation. In the *Need Identification* phase the customer conceptualises the need for a product or service. In the *Product Brokering* and *Merchant Brokering* phases the customer decides which product or service is needed and selects a suitable supplier or service provider. In the *Purchase and Delivery* phase the product is delivered or the service provided, and in the *Product Service and Evaluation* phase the customer advocates his/her satisfaction of the process, products or services provided.

To illustrate the application of RAMASD in this case study, a simple B2B e-commerce model is considered by abstracting from [82] and [53]. The B2B e-commerce model used in this case study includes three phases:

- *Quotation Phase*: In this phase, potential trading parties discover each other and quotations for automotive manufacturing industry parts and supporting services are issued.
- *Auction/Negotiation Phase*: An auction is established by potential buyers of the automotive industry. Subsequently, buyers and sellers negotiate and reach agreements regarding supplying products and providing services. Those agreements are examined by representatives of an appropriate inspection body, for example the Federal Trade Commission, as far as it concerns legal, ethical and social issues, and appropriate action is taken where required.
- *Fulfilment Phase*: In this phase, the contracts agreed in the negotiation phase are executed. For example, the shipping orders of purchased products are submitted to the appropriate departments and the provision of hired services commences. This phase includes all

communication events relevant with gathering customer input about the quality of the products received and the services provided.

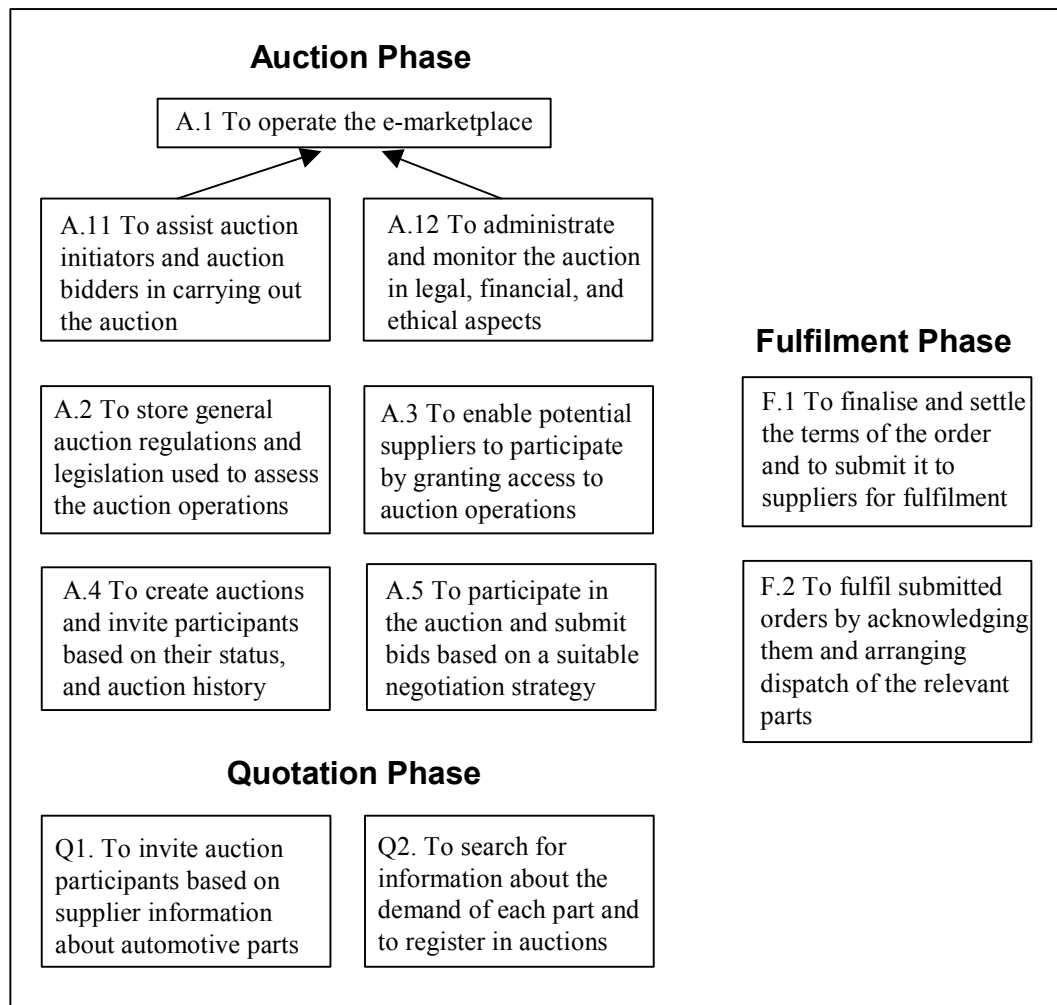


Figure 7.7: Use case goals for an automotive industry B2B exchange case study

The complete automotive industry case study is large and, therefore, for the needs of this chapter, it is assumed that it is only needed to design an ABS to support routine automotive manufacturing supply-chain management tasks. In the example scenario, automotive industry manufacturers first identify their needs for automotive manufacturing parts and services using their proprietary, possibly legacy systems. Subsequently, they search product catalogues and business directories for suitable potential suppliers and service providers. For example, it is common for car manufactures to outsource the manufacturing of seats and exhausts and the re-deployment or disposal of used assets to specialised companies to reduce costs and to increase focus on their main tasks. When a potential buyer identifies some suitable potential suppliers or service providers then they initiate an auction and invites them to participate. Invited trading partners conduct the auction and the potential buyer may accept or reject the outcome based on whether their personal business goals are satisfied. An inspection body representative checks the

auction process and outcome and on receiving approval then the signed contracts are ready for execution. Finally, the suppliers and service providers fulfil their contracts by confirming shipping of the relevant products and starting provision of the relevant services.

7.3.2 Role Identification

In order to model the above system in terms of roles, the first thing to do is to identify the roles involved in the case study example using the role identification technique described in Section 5.2.4. For the purpose of the automotive industry example, three use cases each corresponding to a phase of the simple B2B e-commerce model described in Section 7.3.1 are considered:

- *Trading partner discovery and request for quotation (Quotation Phase)*: This activity involves extensive information exchange among potential trading partners. Each side must sift through large amounts of data for relevant information to make decisions, proposals and counter-proposals. The outcome of this activity is a number of potential suppliers and/or service providers for each potential buyer.
- *Auction Initiation, Negotiation and Monitoring (Auction/Negotiation Phase)*: This involves initiation and establishment of an auction from each potential buyer, negotiation between the trading parties and monitoring of the auction process and results from some external inspection body.
- *Order Fulfilment (Fulfilment Phase)*: In this activity, all interaction regarding execution of contracts, shipment of products and provision of services takes place.

Each use case has a number of high-level goals depicted in Figure 7.7. The behaviour leading to achieving these goals can be modelled by appropriate roles. Hence, the following roles can be identified (Figure 7.8):

1. *Potential_Buyer (goal Q_1)*: This role describes generic behaviour of the automotive industry manufacturers that are interested in purchasing manufacturing parts, sourcing some of their business processes or selecting collaborators for co-design projects regarding sophisticated automotive manufacturing parts. Potential buyers communicate with various potential suppliers or service providers and request quotations and relevant information. A number of suppliers that have submitted attractive quotations are invited to participate in an auction.
2. *Potential_Trader (goal Q_2)*: Potential traders are suppliers or service providers that communicate with potential buyers providing them with quotations and further information. Potential traders also communicate with each other attempting to establish coalitions and submit more attractive offers to potential buyers.

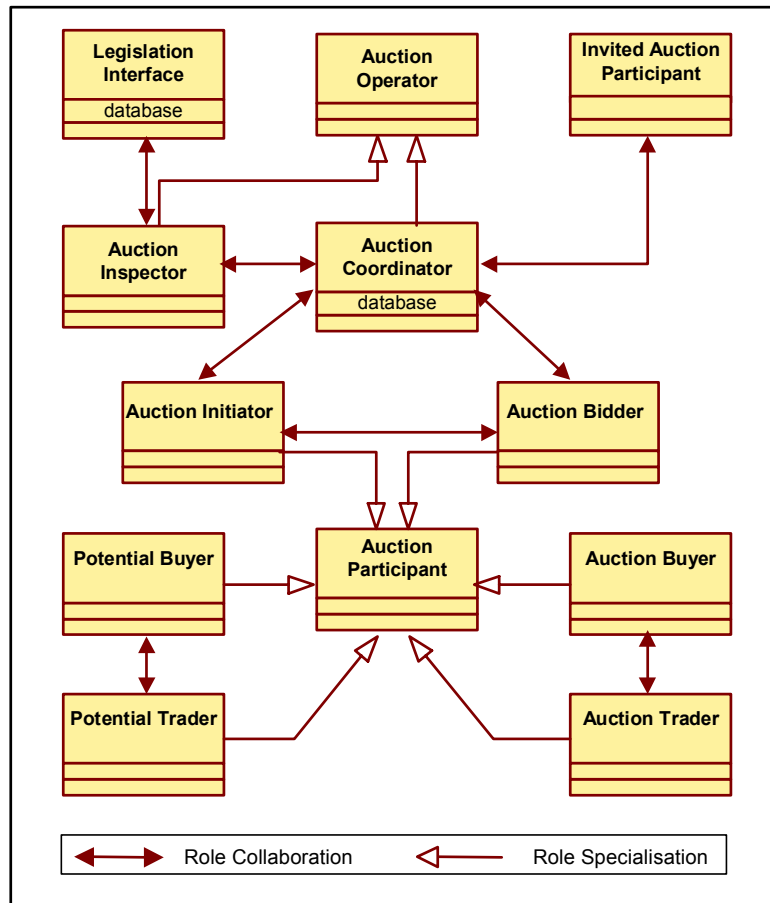


Figure 7.8: Role models for the automotive industry B2B exchange case study

3. *Auction_Operator* (goal A_1): This role describes generic behaviour of the members of the auction operation support groups. An example of this type of behaviour is accessing common auction information including bidding history and trading participants status.
4. *Auction_Coordinator* (goal A_{11}): The *B2B Auction Coordinator* role describes the behaviour required to coordinate the operation of an auction. This includes informing the trading parties about the auction regulations, providing information about the participants' profiles and gathering statistical data of bidding histories. Furthermore, the performance of auction participants and the efficiency of the auction mechanisms is also monitored.
5. *Auction_Inspector* (goal A_{12}): The *Auction_Inspector* role ensures the smooth operation of the auctioning process. The *Auction_Inspector* accesses the auction data gathered by the *Auction_Coordinator* role and verifies that the process followed is legitimate. This is achieved by comparing auction process data with the auction rules and regulations obtained by communicating with the *Legislation_Interface* role, defined below.

6. *Legislation_Interface (goal A₂)*: This role maintains a database of rules and legislation that govern the auction operations. Auction inspectors interact with this role and submit queries regarding auction procedures receiving answers in various data formats.
7. *Invited_Auction_Participant (goal A₃)*: This is a utility role providing access to auction operations to selected suppliers and service providers. This involves authorisation codes to participate in the auction business and access to profiles of other participants and historical auction data.
8. *Auction_Initiator (goal A₄)*: The *Auction Initiator* role is responsible for initiating and running the auction. Its duties include selecting the auction type, bid limits and starting price. Subsequently, it participates in the negotiation with auction bidders by accepting bids and finally establishing a product purchasing or service provision agreement.
9. *Auction_Bidder (goal A₅)*: This role participates in the auction and submits bids aiming to achieve a business contract at a beneficial price.
10. *Auction_Buyer (goal F₁)*: *Auction_Buyers* are active in the order fulfilment phase and they interact with suppliers and service providers to finalise the details of product shipment and start of service provision. They also ensure receipt of products and smooth utilisation of the contracted services by interacting with inventory and proprietary workflow management software.
11. *Auction_Trader (goal F₂)*: This role is also active at the order fulfilment phase and it interacts with buyers to confirm receipt of shipped parts and prompt initiation of agreed services. Its responsibilities include notifying the shipping departments to execute shipment orders, informing the service departments to start provision of the contracted services and interacting with the logistics and accounting departments to ensure that appropriate payment is received.
12. *Auction_Participant*: As noted in [2] it is good practice in role modelling to extract any common behaviour from a number of roles and model it separately in a new role. From the roles identified above, *Auction_Initiator*, *Auction_Bidder*, *Potential_Buyer*, *Potential_Trader*, *Auction_Buyer* and *Auction_Trader* have some behaviour in common. They all represent automotive industry parties that interact within the B2B exchange environment. This common behaviour is modelled by a separate role, which facilitates understanding of the resulting role models and specifying necessary constraints among roles.

7.3.3 Qualitative Modelling of Non-Functional Aspects

In the case study, two non-functional aspects are qualitatively modelled and taken into account when designing the multi-agent system: *security* and *privacy*. The adopted security strategy that is implemented in the designed multi-agent system distributes access to different information sources and to different software agents. Privacy is ensured by intermediation of trading interactions.

7.3.3.1 Security Issues

Creating effective e-business software security strategies and infrastructures is currently one of the biggest challenges in the e-business software industry. IDC predicts that the U.S. information security services market will grow from \$2.8 billion in 1999 to over \$8.2 billion in 2004 [89].

Common B2B software security requirements include identification/authentication of users to enter the system, authorisation to enable them to access the permitted software functionality, user accountability, administration and, most importantly, asset protection. Security strategies typically balance the degree of support to each requirement according to the general policies of the business. For example, higher access privileges to users results in lower software system security.

A security strategy initially requires high level recognition of the business security concerns, which can be described as simple statements. Examples of high level security concerns include monitoring all user activity, ensuring no access to unneeded data and promoting security awareness among employees. Based on high level descriptions of security concerns more specific descriptions of security policies are introduced. Security policies are meant to address security issues when implementing business requirements. Examples of security policies are to use out of band communication when responding to an incident alert, to employ encrypted data exchange techniques, and to maintain a central transaction log server. Security policies lead to specific security actions. For example, disable telnet and ftp in all externally accessible computers, validate html form data both on client and server side and create an extra authorisation level for particularly sensitive and important data.

There have been many approaches to classifying security strategies for possible reuse. For example, a common approach is to apply the *limited view* security strategy discussed by [216]. According to the limited view strategy, users see only what they are allowed to access. Another typical strategy to strengthen the security of a distributed application is to provide a *secure access layer* combining both application and low-level, network security [171, 216].

Based on those two security strategies, it is considered that not only agents should exchange information using secure protocols and over a secure communication medium, but also agents should not have access to information resources relevant to the operation of incompatible roles. The reason is that agents are highly flexible and configurable software components that can alter their behaviour on run-time. For example, the goals of the *Auction_Coordinator* role could lead to attempts to modify an auction legislation database, if it had access to it. In the example considered a very simple security policy for role based access control forbids the access of any agent to more than one database, where a *database* will refer to the set of all data sources relevant to a single role.

7.3.3.2 Privacy Issues

Consumers and organisations that do business over the Internet want assurance that their transactions remain private and no outside parties can access sensitive personal data. This is particularly true in the emerging automotive industry business-to-business models that include vendors and external partners early in the business process, from product design through delivery and support. Competitors sometimes cooperate to complement each other's capabilities. For example, in the defence automotive sector multiple manufacturers collaborate on contracts because of size, complexity and the need for specialised services. Furthermore, one manufacturer may team up with a supplier that is also collaborating with its competitors. For example, an automotive supplier producing seats might be working with several competing auto manufacturers on future designs. As an organisation increases the size of its network, the variety of its markets, inputs and outputs increases [138] — which also increases its needs for privacy.

Although the technology exists to ensure privacy in personal and business communications and data, many companies that acquire private data from customers do not apply the necessary privacy practices. Therefore, software design solutions ensuring privacy among trading parties are required. It has been suggested that to support privacy internet-based software should be based on a centralised data model and that non-public information should be disseminated to interacting parties by a trusted third party [3]. Intermediation has been successfully used in many application domains to enforce privacy, including electronic stock markets [105], manufacturing [181] and mobile workforce management [199].

Intermediation can be modelled by the *mediator role interaction pattern*⁵ (Figure 7.9). This pattern involves three roles:

⁵ More information about the use of the mediator role interaction pattern can be found in “Kendall, E. A. Agent Analysis and Design with Role Models. *Volume 1: Overview*, Martlesham Heath, UK: BT Exact Technologies, (January 1999), unpublished internal BT report”.

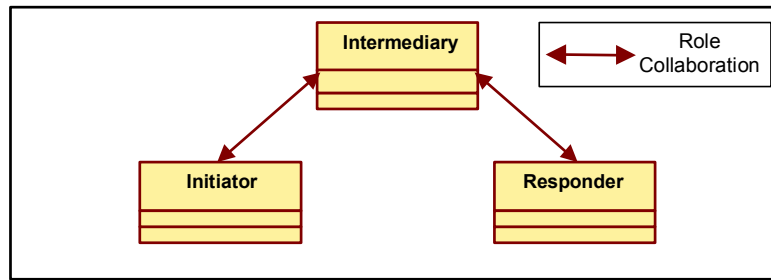


Figure 7.9: The mediator pattern

1. *Initiator*: This role is active in the order fulfilment phase and it interacts with buyers to confirm receipt of shipped parts and prompt initiation of agreed services. Its responsibilities include notifying the shipping departments to execute shipment orders, informing the service departments to start provision of the contracted services and interacting with the logistics and accounting departments to ensure that appropriate payment is received.
2. *Intermediary*: The *Intermediary* role has access to all relevant information of both interacting parties. However, the intermediary does not just filter and selectively communicate information to initiators and responders. In addition, intermediaries can reduce the costs of many information-intensive tasks by integrating customer-based functionality with privacy and security issues,. For example, the intermediary can maintain a database of previous interactions among the same or relevant participants and can provide aggregated results considering any privacy limitations.
3. *Responder*: The *Responder* role is similar to the *Initiator* role with the difference that the *Responder* role responds and continues an interaction that was previously started by the *Initiator* role.

7.3.4 Organisational Settings

There is the requirement in an electronic market place that all transactions are logged for future inspection when needed. This is a general rule of business organisation that needs to be observed⁶. Transactions can be of different types for example, bids, and purchase orders. The transaction logging behaviour can be represented by an organisational role, which in this case

⁶ The difference between organisational requirements and application requirements is subtle as organisational requirements can be considered as application requirements and vice versa. A rule of thumb is to classify requirements as organisational if they could be valid in other software applications the business organisation needs for its operation. Logging transactions is a requirement that can be the case in all software concerning the business operations and hence it can be modelled as an organisational rule.

study is the *Transaction_logger* role. The *Transaction_logger* role simply logs on communication messages to some appropriate database. Hence, each agent in the electronic marketplace that is involved in transactions should be able to play this role. This is modelled with appropriate design constraints as described in Section 7.3.6.

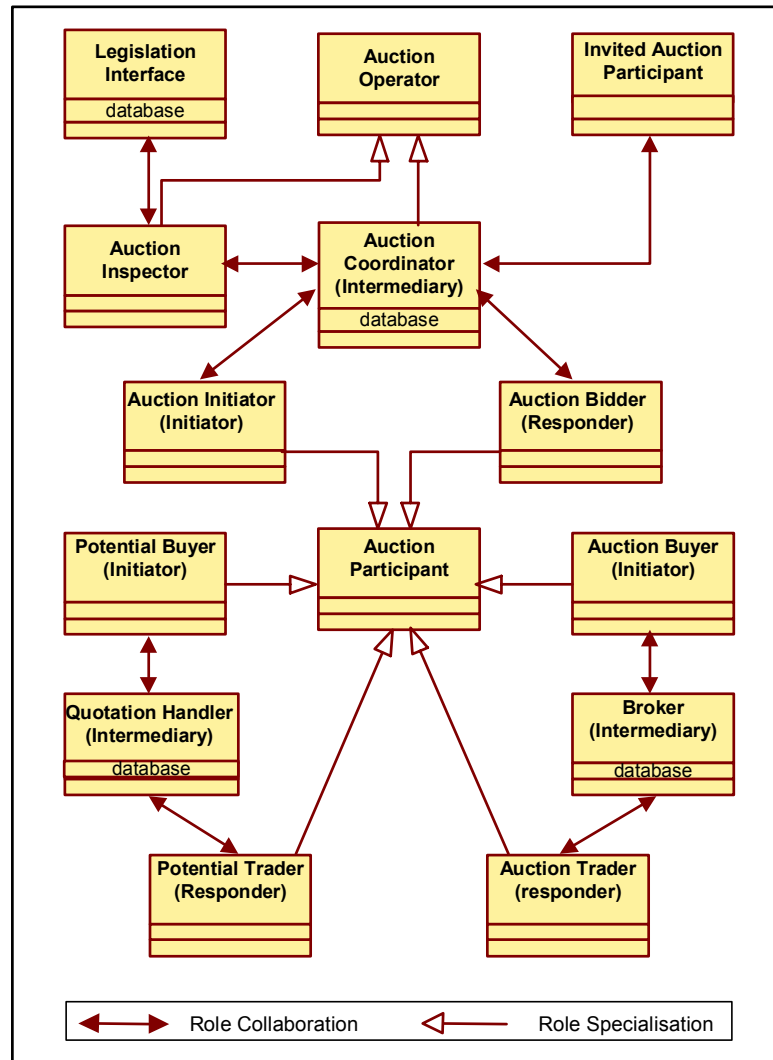


Figure 7.10: Updated role models based on the mediator pattern

7.3.5 Role Composition

When the mediator pattern is considered with the roles identified in Section 7.3.2, role composition takes place. The resulting role models include the mediator pattern. The differences are depicted in Figure 7.10, which describes the resulting role models. Some roles remain the same and some roles are replaced with new roles based on a *Mergeswith* relationship. The new roles are named by combining the names of the roles that contributed to their creation. The role-pairs *Potential_Buyer* - *Potential_Trader*, *Auction_Initiator* - *Auction_Bidder* and *Auction_Trader* - *Auction_Buyer* are replaced as the new roles interact only through

intermediaries. Furthermore, the new roles can inform the intermediaries about their privacy requirements and make use of the utility services, for example statistic analyses, provided by the intermediaries. The *Auction-Coordinator* role is also replaced and the new role acts both as an *Intermediary* and *Auction_Coordinator*.

Two new roles are needed at this stage:

1. *Quotation Handler*: This role provides a means to enhance the communication between potential buyers and potential suppliers. Apart from intermediation, *Quotation_Handler* offers various services to both parties, for example electronic document management and analysis of data gathered throughout similar automotive sourcing requests for quotation. The role has access to a central repository of service sourcing documentation, and product price lists.
2. *Broker*: This role intermediates between buyers and suppliers or service providers in the fulfilment phase. It provides the functionality for many utility tasks including bill of materials, order management, shipping management and returns and status tracking. The role maintains a database of order fulfilment critical information, such as inventory levels, usage history and patterns, receipts and other relevant information to help eliminate excess inventory and premium transportation charges.

7.3.6 Specifying Design Constraints

To illustrate the compositional constraints for the roles identified in Sections 7.3.2 and 7.3.3 and 7.3.5, a number of those constraints is presented written in RCL in Figure 7.11. Role names have been abbreviated for clarity. Some important aspects of the RCL specification of those constraints are described in more detail here.

As can be seen from Figure 7.11, the *Legislation_Interface*, *Auction_Coordinator*, *Quotation_Handler* and *Broker* roles have associated the integer performance variable *database*. This represents a proportion of the amount of actual storage space required by those roles.

To model the requirements discussed in the previous sections, several containment constraints are needed. For example, an *a_coordinator* contains the *a_operator* role written `in(a_coordinator, a_operator)`. The number of exclusion constraints, for example, an agent that is *Auction_Coordinator* cannot also be *Auction_Inspector*, is also significant. The *Excludes* relation is also used to specify that interacting roles in the request for quotation, auction negotiation and order fulfilment processes should be played by different agents.

```

/* ROLE DEFINITIONS */
Role a_operator, a_participant,
    a_inspector, a_coordinator,
    a_initiator, a_initiator_i,
    a_bidder, a_bidder_r,
    p_buyer, p_buyer_i,
    p_trader, p_trader_r,
    a_buyer, a_buyer_i,
    a_trader, a_trader_r,
    ia_participant;

merge(a_coordinator, intermediary,
      a_coordinator_int);

merge(p_buyer, initiator, p_buyer_i);
merge(a_initiator, initiator,
      a_initiator_i);
merge(a_buyer, initiator, a_buyer_i);

merge(p_trader, responder, p_trader_r);
merge(a_bidder, responder, a_bidder_r);
merge(a_trader, responder, a_trader_r);

Role l_interface, a_coordinator,
    q_handler, broker {
    int database;
}

l_interface.database = 1;
a_coordinator.database = 1;
q_handler.database = 1;
broker.database = 1;

not(a_participant, a_operator);
not(l_interface, a_coordinator);
not(l_interface, a_participant);

not(a_coordinator, a_inspector);
not(a_coordinator, a_participant);
not(q_handler, a_participant);
not(q_handler, a_inspector);
not(broker, a_participant);
not(broker, a_inspector);

/* ROLE CONSTRAINTS */

in(a_coordinator, a_operator);
in(a_inspector, a_operator);
in(a_initiator, a_participant);
in(a_bidder, a_participant);
in(p_buyer, a_participant);
in(p_trader, a_participant);
in(a_buyer, a_participant);
in(a_trader, a_participant);

not(a_initiator, a_bidder);
not(p_buyer, p_trader);
not(a_buyer, a_trader);

/* GENERAL CONSTRAINTS */

Constraint Y {
    forall a:Agent {
        a.database <= 1
    }
}

```

Figure 7.11: Compositional constraints for the B2B exchange case study

In order for an agent to be an *Auction_Bidder* and participate in an auction, it must have been previously invited by *Auction_Initiator*. This is modelled using the *Requires* relation to specify that the *Auction_Bidder* role must be played together with the *Invited_Auction_Participant* role. Furthermore, the changes in the behaviour of roles when the mediator pattern is applied are modelled using the *Mergeswith* relation. For example, an *Auction_Coordinator* merges with the *Intermediary* role resulting in the *Auction_Coordinator_Intermediary* role, which combines the behaviour of both *Auction_Coordinator* and *Intermediary* roles.

For security reasons, neither *Auction_Coordinator* nor *Auction_Participant* can coexist with the *Legislation_Interface* role. For the same reason, *Auction_Coordinator* cannot coexist with

Auction_Inspector and *Auction_Coordinator*, *Quotation_Handler* and *Broker* cannot coexist with *Auction_Participant*. Those constraints are also specified using the *Excludes* relation.

Further constraints are specified on agent and role characteristics. For example, to increase security it would be desirable for the agents to have access to not more than one information source. This is modelled by constraining the *database* performance variable to be at most one for all agent types. The value of the *database* variable of an agent type is equal to the sum of the values of the *database* variables of the roles the agent plays. In this calculation, only roles that have the *database* variable defined are considered. For example, assuming that an agent plays the *Quotation_Handler*, the *Auction_Coordinator* and the *Auction_Operator* roles, then the agent is automatically associated with the performance variable *database* since at least one of the roles it plays is associated with this variable.

The value of the agent *database* variable would be two since the values of the *Auction_Coordinator* and *Quotation_Handler* *database* variables are one each and the *Auction_Operator* role is not associated with a *database* variable.

7.3.7 Role Allocation Results

In this section an application of the algorithm to the RCL specification described in Section 6.5 is presented. The results of the role allocation are summarised in Figure 7.12. It is assumed that the algorithm starts randomly from the `merge(a_coordinator, intermediary, a_coordinator_int)` constraint. Those three roles are allocated to the first agent type. Since `a_coordinator` contains `a_operator`, the `a_operator` role is also allocated to this agent type. All constraints involving the allocated roles so far are satisfied. Subsequently, a check of the general constraint `a.database <= 1` is done and it is successful since `a.database = a_coordinator_int.database = 1`. The next step is to attempt to allocate roles from the `merge(p_buyer, initiator, p_buyer_i)` constraint. The role `p_buyer` contains `a_participant` and hence an attempt to also allocate `a_participant` is made. However, `a_coordinator` excludes `a_participant` and therefore this allocation attempt fails. Subsequent attempts to allocate roles participating in the remaining merging constraints fail for the same reason.

The algorithm then proceeds to create a second agent type. In the second agent type, roles involved in the next three merging constraints are successfully allocated. Since `p_buyer`, `a_initiator` and `a_buyer` contain `a_participant`, the `a_participant` role is also allocated to this agent type and the general constraint is satisfied. The series of successful allocation steps is interrupted when the algorithm attempts to allocate roles from the `merge(p_trader, responder, p_trader_r)` constraint. The reason is that a

p_buyer cannot be a p_trader in the auction type considered, which is specified in Figure 7.11 using an *Excludes* relation.

The next step is to create a third agent type where the roles involved in the remaining merging constraints are successfully allocated. Since p_trader, a_bidder and a_trader contain a_participant, the a_participant role is also allocated to this agent type. The ia_participant role is also allocated because a_bidder requires ia_participant. As there are no merging constraints left, the algorithm then attempts to allocate the remaining roles. This attempt fails for this agent type since q_handler, broker, a_operator and l_interface cannot coexist with a_participant, as specified in Figure 7.12.

A fourth agent type is therefore created and the q_handler role is randomly allocated to it. The algorithm fails to allocate l_interface to this agent type since, as explicitly specified in Figure 7, q_handler cannot coexist with l_interface. The algorithm also fails to allocate a_inspector since in that case it would have to also allocate a_operator, contained by a_inspector, which is excluded by q_handler. Finally, broker cannot be allocated since in that case the agent database value would be two and the general constraint would be violated.

Agent Type 1		Agent Type 2		Agent Type 3	
has	plays	has	plays	has	plays
a_coord intermediary a_coord_int	a_coord_int	p_buyer a_initiator a_buyer initiator p_buyer_i a_initiator_i a_buyer_i a_partcpnt	p_buyer_i a_initiator_i a_buyer_i	p_trader a_bidder a_trader responder p_trader_r a_bidder_r a_trader_r a_partcpnt ia_partcpnt	p_trader_r a_bidder_r a_trader_r
database	1	database	0	database	0

Agent Type 4		Agent Type 5		Agent Type 5	
has	plays	has	plays	has	plays
q_handler	q_handler	l_interface a_inspector a_operator	l_interface a_inspector	broker	broker
database	1	database	1	database	1

Figure 7.12: Agent types for the B2B exchange case study

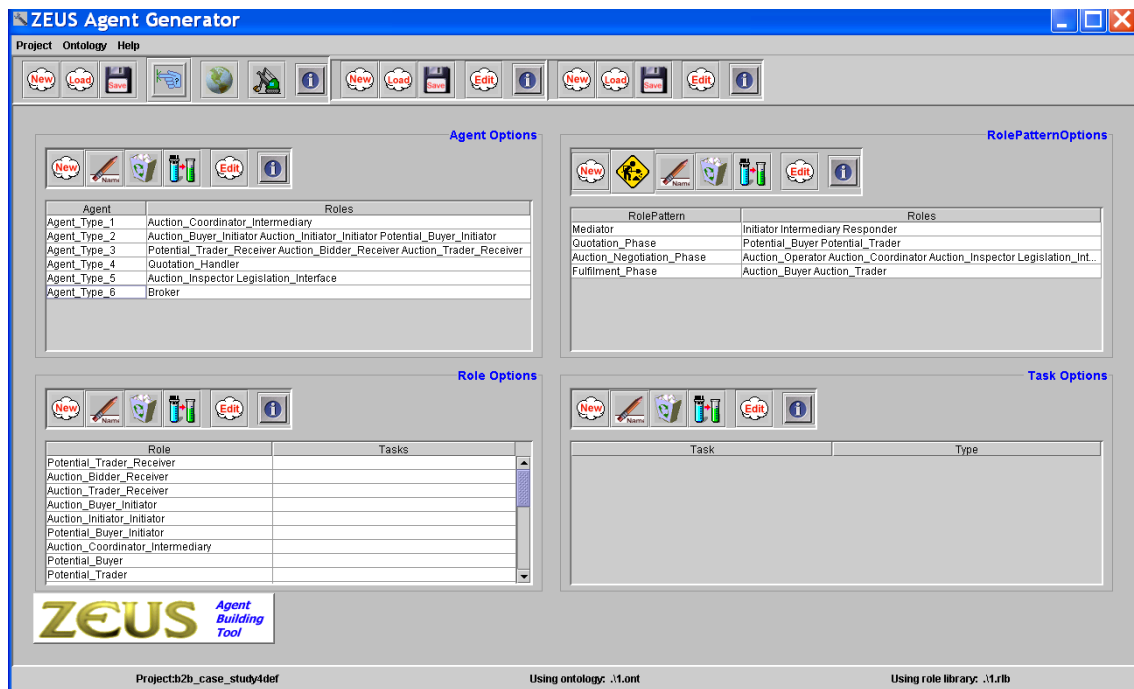


Figure 7.13: Snapshot of the extended Zeus toolkit for the B2B exchange case study

This leads to creating a fifth agent type where the `l_interface` and `a_inspector` roles are successfully allocated. Since `a_inspector` contains `a_operator`, the `a_operator` role is also allocated to this agent type. However, the `broker` role cannot be allocated since it cannot coexist with `a_inspector` as specified in Figure 7.12.

Finally, a sixth agent type containing the remaining role `broker` is created and the algorithm ends with success.

A snapshot of the role allocation using the extended Zeus toolkit is shown in Figure 7.13.

7.4 Summary – Conclusions

In this chapter the RAMASD method and its applicability were demonstrated in two case studies drawn from real world applications, Mobile Workforce Support and B2B Electronic marketplace management. The first case study included the demonstration of how RAMASD handles quantitative modelling of non-functional aspects and design heuristics and the second one provided an example of how non-functional aspects and organisational aspects can be handled qualitatively.

Chapter 8

Evaluation of RAMASD

This chapter assesses the value of RAMASD with respect to design complexity, highlighting its advantages in comparison with other ABS design methods. It discusses how RAMASD addresses the open issues identified in Section 3.3, and reports on lessons learned from applying RAMASD in the case studies discussed in Chapter 7. It then uses the evaluation framework proposed in Section 3.1 to compare RAMASD with a representative selection of ABS design methods. Finally, it provides a detailed analysis of how RAMASD and a representative ABS design method, Gaia, differ in dealing with the mobile workforce case study.

The contents of this chapter are as follows. The approach followed in selecting the case studies and the evaluation methodology applied are discussed in Section 8.1. Subsequently, a framework-based comparison between RAMASD and existing ABS design methods is described in Section 8.2. The results of the detailed comparison between RAMASD and Gaia are described in Section 8.3, followed by a discussion regarding the novel aspects and the applicability of RAMASD in real world applications in Section 8.4. Finally, Section 8.5 concludes the chapter.

8.1 Selecting an Evaluation Approach

Evaluation of software engineering methods can be done based on descriptive analysis or experimentation in a manner similar to the one applied in general science. In this thesis, a two-stage approach has been selected. At the first stage, the value of RAMASD as compared with other ABS design methods has been assessed using an evaluation framework. At the second stage, RAMASD is directly compared with Gaia, a representative ABS design method. In both stages the comparisons are based on the case studies described in Chapter 7. The case studies were selected from business domains and the test scenarios were chosen so that they would cover all issues (design heuristics, organisational settings and both qualitative and quantitative non functional aspects) that RAMASD supports.

8.1.1 Approaches to Evaluating Software Engineering Methods

A software engineering method can be evaluated using either descriptive evaluation or evaluation by experimentation [223]. Descriptive evaluation involves assessing the method based on descriptions of its characteristics. Evaluation by experimentation involves examining results from experiments where the method has been applied to develop software products.

Descriptive evaluation approaches involve arguing for or against certain characteristics of the evaluated method without actually applying it. This is useful to identify weaknesses in existing software engineering methods, for example when desirable features are not supported. Descriptive evaluation can be focused on interviewing the software engineers that use the method, such as in [195], on examining the characteristics of the method based on an evaluation framework (such as in [137]) or on carrying out uniform comparisons of the evaluated method with other methods using a meta-model (such as in [86]).

Evaluation by experimentation refers to applying the method and the subsequent collection of data on either the process followed or on the software artefact produced. Experimentation can be *observational*, *historical* or *controlled* [223]. Observational experimentation methods collect relevant data as the software projects develop, whilst historical methods collect data from projects that have already been completed. Controlled methods collect multiple instances of the same data and compare them using qualitative and statistical methods. For example, controlled experimentation data can be collected by developing a software product both with and without using the evaluated method.

8.1.2 Evaluating RAMASD

The value of RAMASD was assessed by a combination of descriptive and experimental evaluation. This involved using an evaluation framework to carry out a comparison between RAMASD and a number of ABS design methods and performing a detailed comparison between RAMASD and Gaia, a representative ABS design method. Both evaluation tasks have been based on the results of the case studies described in Chapter 7.

Using an evaluation framework to compare RAMASD with other similar methods was considered as the most suitable alternative. It was not practical to evaluate it based on interviews of software engineering professionals since RAMASD is still at the experimental stage. Furthermore, RAMASD involves some unique characteristics, such as the role algebra, which make it difficult to represent by a generalised meta-model. Therefore, it was decided to use the evaluation framework introduced in Section 3.1 to carry out a framework-based evaluation of RAMASD. Framework-based evaluation has the important advantage of allowing the evaluation of a number of methods against multiple criteria, which was considered particularly suitable for comparing RAMASD with other methods with respect to reducing design complexity. This is further discussed in Section 8.2.2.

The value of RAMASD was demonstrated by a detailed comparison between RAMASD and Gaia, a well-known ABS design method was carried out in the context of the mobile workforce case study. This enabled demonstrating the RAMASD value by highlighting the Gaia weaknesses when applied to this case study.

8.1.3 Selecting Case Studies and Test Scenarios

RAMASD targets primarily business-oriented application domains and this was the basis for selecting the case studies to use for the evaluation. Furthermore, the test scenarios were selected so that all issues currently supported by RAMASD would be demonstrated.

The main use of agent technology in business domains is using agents as personal assistants to users, augmenting their abilities and acting on their behalf. Therefore, the first case study selected was one involving supporting mobile workforce where agents primarily aim to assist users and extend their abilities. In this case, there is frequent interaction between agents and their users. The second case study involves an electronic marketplace where agents aim to act on behalf of their users interacting therefore less frequently with them.

Furthermore, the test scenarios were selected to demonstrate the main features of RAMASD in the way of reducing design complexity: support for organisational aspects, design heuristics and both qualitative and quantitative non-functional aspects.

8.2 Framework-Based Evaluation

RAMASD was developed to address the weaknesses of ABS design methods concerning design complexity. These weaknesses were discussed in Chapter 3 using a specially constructed evaluation framework. This evaluation framework is now applied to (a) shape a discussion of how RAMASD addresses those weaknesses using results from Chapter 7 and (b) to compare RAMASD against the rest of the ABS design methods.

8.2.1 Main Features of RAMASD

There is a consensus in the literature that reducing design complexity involves enabling the designer to work at a high level of abstraction and to semi-automate the design process. RAMASD supports designers in both aspects by encapsulating complex behaviour in the role definition, by formalising role relations in the role algebra and by providing a semi-automatic design process based on the synthesis concept. The main features of RAMASD, which address the challenges identified in Section 3.3 are discussed here.

Design heuristics: RAMASD supports design heuristics by representing them as constraints on roles and between agent and role characteristics. In the example considered in Section 7.2.3, the point of interaction heuristic was applied to require that both travel information and knowledge expertise retrieval interactions should be carried out by the same agent. This was specified as a requirement constraint between the roles *TravelManager* and *KnowledgeFinder*.

Organisational settings: RAMASD addresses this issue by explicitly modelling and considering organisational settings using organisational roles, which can then be constrained

using the role algebra. In most cases, organisational roles will be merging with application roles resulting in different behaviours for the agents playing them. This has been illustrated in the example given in Section 5.5.3. However, there can be cases where organisational roles will simply constrain the existence of other roles. This has been demonstrated in the COVISINT case study (Section 7.3.2) where the *Transaction_Logger* role is a requirement for all agents playing the *Auction_Participant* role. It is also possible to model organisational rules as constraints on agent and role characteristics in a manner similar to non-functional qualities; however, explicitly using organisational roles results to easier to understand role models and therefore it should be preferred.

Collective behaviour: RAMASD addresses this issue by adopting a role modelling approach where collective behaviour is represented by role models. This is similar to what is done in other areas, for example pattern-oriented programming. However, RAMASD goes one step further since it allows for automatic composition of role models based on constraints described in the role algebra. To the author's knowledge this has not been applied in ABSs design elsewhere. Representing collective behaviour with role models has been illustrated in both case studies described in Chapter 7.

Non-functional aspects: Non-functional aspects are supported in RAMASD in both quantitative and qualitative manner. This has been illustrated by the memory requirement in the mobile workforce case study and by the privacy and security requirements in the COVICINT case study. To the author's knowledge, this is the only method for designing ABSs where both quantitative and qualitative non-functional aspects are considered in the same design model.

Automating the design process: RAMASD automates certain steps in the design process, for example using the algorithm introduced in Chapter 6. The algorithm was demonstrated step by step in the case studies design results (Sections 7.2.4 and 7.3.7 respectively). RAMASD attempts to mitigate the intractability problem in automatic software design using a process based on the synthesis concept. Part of the process steps are carried out by the designer and part are carried out automatically. Each role model allocation is considered as a separate synthesis sub-problem and the allocation of all role models is the solution to the overall synthesis problem⁷. The synthesis concept has been applied in many areas of software engineering, for example in the design of embedded systems [164]. However, to the author's knowledge it has never been previously applied to role allocation for the design of ABSs. Using synthesis reduces design complexity since the designer has to handle fewer design issues in an ad-hoc manner.

⁷ As mentioned in Section 5.3, the solution to the overall synthesis problem consists of the combination of the solutions to synthesis sub-problems.

Working on different abstraction levels: The ABS designer can work at three different abstraction levels: The *role characteristic level* where he specifies the characteristics of new roles and introduces constraints between agent and role characteristics, the *role level* where he specifies new roles and introduces inter-role constraints and the *role model level* where he selects new role models to reuse from the role model library. In this way, after the designer has designed a number of ABSs and has obtained access to predefined role libraries he will be mostly operating at the role model level having to manually handle minimum design complexity. In the case studies in Chapter 7, all three abstraction levels were demonstrated.

Similar approaches based on systematically gluing conceptual models together exist in the area of pattern languages [54, 215]. However, to the author's knowledge none of those approaches allows representation of complex behaviour, for example, non-functional aspects and organisational settings. RAMASD is unique in this respect.

Another issue that is worth highlighting is that it is also possible to reuse existing ABS architectures that have been found suitable for some application domains, for example the PROSA reference architecture [213], by representing them as role models, stored in the role model library together with application and organisational role models, and requiring them to be included in the role allocation process. In this way, known design solutions can be combined with application functionality without the designer having to go into details about how this can be achieved at the design level.

Based on the above discussion, it can be argued that RAMASD adequately addresses the issues raised in Chapter 3.

8.2.2 Comparing RAMASD With Other Methods

RAMASD has addressed all issues raised in Section 3.2 and hence it can be considered superior to existing ABS design methods in regards to reducing design complexity. To illustrate this, the evaluation framework proposed in Section 3.1 is used. Based on the discussion carried out in Section 8.2.1, it can be argued that RAMASD performs well with respect to all perspectives of the evaluation framework. The comparison results are summarised in Table 8.1.

Regarding the *Concepts* perspective, RAMASD generally does not target a specific agent architecture nor does it produce specific agent types. In the context of this thesis, RAMASD has been integrated with the Zeus agent building toolkit and hence the implemented tool produces only Zeus agents bounded by the Zeus agent architecture. However, RAMASD could be very well implemented in other toolkits, for example JADE [13] without any need to modify the method itself. Furthermore, RAMASD obviously has the design phase of the ABS engineering life-cycle in its scope as it was developed to be a design method. Finally, as demonstrated in the mobile workforce case study (Section 7.2.3), RAMASD provides adequate support for design

heuristics which can be represented by appropriate design constraints.

		RAPPID	DESIRE	Gaia	MESSAGE	Tropos	Zeus	KARMA	RAMASD
Concepts	Concept definition	⋈	◇	×	×	◇	◇	×	×
	Design in scope	–	√	√	–	√	√	√	√
	Heuristics support	–	–	–	–	–	–	√	√
Models	Organisational settings	–	–	–	–	–	√	√	√
	Collective behaviour	–	–	–	–	–	√	√	√
	Non-functional aspects	–	–	–	–	√	–	–	√
Process	Design perspective	↓	↓	↓	↕	↓	↑	↓	↕
	Support for reuse	–	√	–	–	–	√	–	√
	Design automation	–	–	–	–	–	–	√	√
Pragmatics	Generality	○	∅	∅	⊕	⊕	∅	⊕	⊕
	Complexity handling	–	√	–	–	–	–	√	√
	Tool support	–	√	–	√	–	√	√	√

Legend

○ - low	⋈ - limited	↑ - bottom-up	√ - yes
∅ - medium	◇ - bounded	↓ - top-down	– - no
⊕ - high	×	↕ - both	

Table 8.1: Comparing RAMASD with other ABS design methods

As far as it concerns the *Models* perspective, RAMASD does support organisational settings as first class design constructs by means of organisational role models and constraints on organisational role characteristics. Role models are also used for the representation of collective behaviour. Furthermore, non-functional aspects are supported in both qualitative and quantitative manner using non-functional role models and constraints on role characteristics.

In the *Process* perspective, RAMASD allows designing in both bottom-up and top-down fashion. Bottom-up design in RAMASD is similar to the one done in [111]. A number of role models are selected and design progresses upwards. The extension done from RAMASD in this respect is that the role algebra allows automatic composition of role models. The role algebra also enables top-down design since high level organisational settings can be represented by role models and seamlessly combined with other role models using the role algebra. This is not

possible in existing ABS design methods. Furthermore, RAMASD explicitly supports reuse by means of role interaction patterns. In particular, enabling systematic reuse of design knowledge based on the role algebra is one of the main innovations of RAMASD. Finally, the role algebra is the basis for automating certain design steps.

Regarding the *Pragmatics* perspective, RAMASD is characterised as having high generality since it can be used to design all currently known types of ABSs. As discussed in Section 8.2.1 RAMASD enables work at different levels of abstraction. Finally, RAMASD has tool support currently in the form of integration in the Zeus agent building toolkit.

Consequently, RAMASD supports all aspects considered in the evaluation framework of Section 3.1 and hence RAMASD can be considered superior in that respect to the other methods used in the comparison.

8.3 Comparison of RAMASD and Gaia

The previous section demonstrates that RAMASD is superior to current ABS design methods in regards to reducing design complexity. In this section, this is exemplified in a greater level of detail by highlighting possible drawbacks of applying Gaia, a baseline ABS engineering method, to the case studies examples described in Chapter 7. Gaia was selected because (a) it is informal and uses role modelling, (b) It has been applied to a considerable number of research projects and (c) it has been used as a base line method in similar assessments, for instance [222].

8.3.1 Overview of Gaia

As discussed in more detail in Appendix A.3, Gaia [209] involves two analysis models, role model and interaction model, and three design models, agent model, services model and acquaintance model. In this section, the steps required to design an ABS using Gaia are briefly described.

Role identification: To evaluate how Gaia supports the design of ABSs, it is required to consider how roles are identified in the application domain. In Gaia, roles are viewed as abstract descriptions of the agents expected functions in the ABS. Therefore, identifying the main functions in a business system is the basis for role identification. In particular, Gaia considers that in business application domains there is often an one-to-one mapping between departments and roles. Roles in a system will typically correspond to:

- Individuals, either within an organisation or acting independently,
- Departments within the organisation; or
- Organisations themselves.

Based on the above all actors in the mobile workforce case study example correspond to distinct roles. This is further discussed in Section 8.3.2.

Role specification: For each of the identified roles, the associated permissions, protocols and responsibilities need to be specified. For example, protocols in Gaia refer to the patterns of interaction that occur in the system between the various roles. Along this line, a protocol may correspond to the interaction of an agent playing the role of Buyer, submitting bids to another agent in the role of Seller.

Role interactions: After selecting roles and specifying the role characteristics then role interactions are captured in Gaia in the interaction model. The interaction model follows very naturally from the definition of the roles. It basically amounts to specifying which protocols involve which role pairings and what information is exchanged during the execution of the protocol.

Assigning roles to agents: Having completed the analysis phase, the first step of the design phase of Gaia is to assign roles to agents. This is done by the designer in an ad-hoc manner.

Creating the service model: Agent services in Gaia are functions the agents can perform and they can be based on responsibilities, activities and permissions of more than one roles that agents play. The service model in Gaia specifies which services the agents must implement to enable all the roles the agent has to play to be fulfilled. This amounts to transforming the abstract activities that the roles have to perform (as identified in the analysis phase) into more coherent blocks of computational activity. This is done manually by the designer. In the current version of RAMASD there is no notion of agent services.

Creating the acquaintance model: The last design model in Gaia is the acquaintance model. This model simply identifies the communication pathways that exist between agents, provides a check of whether the structure of the interactions in the system are poorly organised. When the agent organisation closely resembles its real world counterpart, as in this example, then there are no obvious bottlenecks.

8.3.2 Applying Gaia in the Mobile Workforce Case Study

In this section, the Gaia method is applied to the example concerning the mobile workforce case study which was described in Chapter 7. The aim of this exercise is to provide more concrete examples of the Gaia limitations as compared to RAMASD with respect to design complexity.

Role Schema:	REPAIR_WORKER
Description:	Describes the behaviour of field engineers
Protocols and Activities	<u>Pull_repair_tasks</u> , negotiate_work_tasks, schedule_work_tasks, provide_mentoring_services, <u>receive_work_practice_info</u> , searches_for_travel_info, searches_for_expert_knowledge
Permissions	Reads submitted repair task requests from the workpool database Reads travel information from the Travel Information database Reads expertise knowledge from expertise knowledge base
Responsibilities	
Liveness	REPAIR_WORKER = SCHEDULE_REPAIR PERFORM_REPAIR OTHER SCHEDULE_REPAIR = { <u>Pull_repair_tasks</u> , negotiate_work_tasks, schedule_work_tasks} ^ω PERFORM_REPAIR = {execute_work_tasks} OTHER = {provide_mentoring_services <u>receive_work_practice_info</u> }
Safety	Tasks_pulled = 0 \Rightarrow Tasks_scheduled = 0 Memory \geq 4

Table 8.2: The role schema for the REPAIR_WORKER role

Based on the role identification guidelines applicable in Gaia (described in the previous section) the behaviours in the the mobile workforce case study can be modelled with five roles⁸: REPAIR_WORKER, MANAGER, CUSTOMER_HANDLER, TRAVEL_DEPT and EXPERTISE_KNOWLEDGE. The first three roles correspond to persons in the business organisation and the last two correspond to departments, the Transportation & Logistics and the Education & Development departments. The details of the role schemata for the above roles are summarised in Table 8.2, Table 8.3 and Table 8.4. The specification of the role characteristics has been done using the Gaia method and notation as described in [209].

Applying Gaia to the mobile workforce case study results to identifying less roles than are identified when using RAMASD. Gaia roles are of higher granularity. For example, the REPAIR_WORKER role assumes the behaviours of the *Coordinator*, *Employee*, *Mentor*,

⁸ Roles names in Gaia are written in capital letters. This convention is also followed in this section. Roles identified using Gaia are written in capitals and roles identified in RAMASD throughout this thesis are written in italics.

TravelInfoFinder and *KnowledgeFinder* roles. The MANAGER role also includes the behaviour of the *Brulebase* role. Hence, to accurately represent the system behaviour, Gaia roles have to be associated with sophisticated responsibilities, activities and permissions are assigned to the Gaia roles. The memory requirements of each role can be represented as Safety constraints.

According to the Gaia method, it is most likely that each role corresponds to an agent type. It is up to the designer to change this correspondence, for example to assign more than one role to an agent type aiming to achieve better run-time performance, but this has to be done in an ad-hoc manner, resulting to increased design complexity. For the case study example, a standard role allocation is to assume that each role identified using Gaia corresponds to a separate agent type.

Role Schema:	MANAGER
Description:	It represents the behaviour of the team manager including confirming task allocation, monitoring work and ensuring that business rules are followed.
Protocols and Activities	Monitor_task_execution, confirm_task_allocation, receive_work_practice_info, <u>update_business_rule_data</u>
Permissions	Reads/writes business rules from the business rule database
Responsibilities	
Liveness	MANAGER = {Monitor_task_execution confirm_task_allocation <u>receive_work_practice_info</u> <u>update_business_rule_data</u> }
Safety	Memory ≥ 2

Role Schema:	CUSTOMER_HANDLER
Description:	Receives a repair request from the customer and interacts with other roles to arrange for a field engineer to visit customer and carry out the repair
Protocols and Activities	<u>Collect_repair_requests</u> , <u>Update_workpool_data</u>
Permissions	Writes submitted repair task requests to the workpool database
Responsibilities	
Liveness	CUSTOMER_HANDLER = {Collect_repair_requests <u>update_workpool_data</u> }
Safety	Memory ≥ 4

Table 8.3: Role schemata for the MANAGER and CUSTOMER_HANDLER roles

Role Schema:	TRAVEL_DEPT
Description:	Store travel information from various resources, i.e. GPS and Traffic databases
Protocols and Activities	Update_travel_data
Permissions	Writes travel information in the travel database
Responsibilities	
Liveness	TRAVEL_DEPT = { <u>update_travel_data</u> }
Safety	Memory ≥ 2

Role Schema:	EXPERTISE_KNOWLEDGE
Description:	Maintains and manages a database of expertise about telephone repair tasks
Protocols and Activities	<u>Update_expertise_knowledge_data</u>
Permissions	Writes submitted expertise knowledge about repair tasks requests to the expertise knowledge base.
Responsibilities	
Liveness	EXPERTISE_KNOWLEDGE = { <u>Update_expertise_knowledge_data</u> }
Safety	Memory ≥ 2

Table 8.4: Role schemata for the TRAVEL_DEPT and EXPERTISE_KNOWLEDGE roles

8.3.3 Limitations of Gaia

The discussion carried out in Section 8.3.1 and the application of Gaia to the mobile workforce case study described in the Section 8.3.2 highlight a number of Gaia weaknesses which result in increased design complexity:

1. *Insufficient role identification:* Role identification in Gaia does not cover all possibilities of representing behaviours using roles. For example, the Gaia view of roles does not consider roles being played by machines, something which often occurs. Furthermore, there is no indication as to how such roles could be identified by the designer, for example how to identify possible roles associated with an individual in an organisation. This requires the designer to identify roles in an ad-hoc manner involving high complexity.
2. *High role granularity:* Role identification in Gaia results in roles of higher granularity than that in RAMASD. This reduces the possibility of reusing Gaia roles in other applications

and hence increases design complexity. Furthermore, as Gaia roles correspond mainly to individuals in organisations the same role is more likely to interact with different roles in different application contexts and hence the need for explicit specification of new interaction protocols is higher. This results in high design complexity for the designers.

3. *Low model reusability*: Gaia models are more difficult to be reused than those of RAMASD. For example, a Gaia interaction model includes all roles involved in a particular application. A different application is unlikely to include exactly the same interacting roles and hence the existing interaction model cannot be reused. Having to redefine conceptual models for each ABS design increases design complexity.
4. *Low abstraction level*: RAMASD allows designers to reason at the role model level while Gaia only allows work at the role level. For example, the *Travel Management* role model described in Section 7.2.2 can be reused without explicitly referring to the characteristics of the *TravelManager* and *TravelInfoBase* roles it includes. In RAMASD, focusing on role characteristics would be needed only when roles had to be customised to better represent particular application requirements. In contrast, Gaia requires the designer to explicitly specify certain role characteristics, such as role interactions, even when the same role is used in the design of more than one ABS. For example, the TRAVEL_DEPT role is likely to interact with different roles when reused in different applications since it represents the generic function of travel information provision. This is another factor that increases the Gaia design complexity.
5. *Lack of support for role specialisation*: Role modelling in Gaia does not support specialisation/inheritance. In the mobile workforce case study the behaviour of the REPAIR_WORKER role overlaps to a certain extent with the behaviour of the MANAGER role. For example, they both have the activity receive_work_practice_info, which refers to receiving information about common work practices and business news. Having to describe roles from the beginning even when part of the role behaviour occurs in more than one roles increases design complexity.
6. *Lack of definition of role relationships*: The REPAIR_WORKER role is an implicit merging of the *Coordinator*, *Travel_Manager* and *Knowledge_Finder* roles identified in Section 7.2.2 using RAMASD. The designer is therefore required to reason about this merging in an ad-hoc manner. Any increase in the memory required to store intermediate results as a result of this merging needs to be implicitly taken into account. In this example, this is done by specifying it as a Safety constraint for the REPAIR_WORKER role schema.
7. *Lack of support for design heuristics*: The point of interaction heuristic, which in Section 7.2.2 requires *Travel_Manager* to be collocated with *Knowledge_Finder*, cannot be

explicitly observed when applying Gaia. In the role modelling done in this section, it happens that the point of interaction heuristic is implicitly observed since the behaviours corresponding to the *Travel_Manager* and *Knowledge_Finder* roles are included in the REPAIR_WORKER role. In general, however, this is not the case and the designer has to apply design heuristics in an ad-hoc manner.

8. *Inefficient support for non-functional aspects:* As for the non-functional aspect of memory, it is taken into account in the design of the agent types only in an ad-hoc manner. For example, the memory of the REPAIR_WORKER role should be manually calculated to be equals to 4 (see Table 8.4). However, this would prevent the designer from selecting a feasible role allocation in the design stage. This is because the requirement that the memory of each agent should be less than or equal to 2 can not be satisfied in any role allocation. After realising this problem, the designer would be expected to manually modify the identified roles, for example to represent the REPAIR_WORKER behaviour using two roles, and reattempt the design. This iteration is similar to the iterations that can occur when applying RAMASD. However, the main difference is that there is no explicit way in Gaia to specify design constraints. The safety and liveness constraints attached to roles in Gaia aim to assist in the specification of the behaviour represented by roles and are used in the specification of the agent services once roles have been allocated to agent types. However, there is no systematic way to specify constraints that would drive design decisions. The designer must realise, represent and apply design constraints in an ad-hoc manner. Obviously, this increases the complexity that the designer must handle.
9. *Lack of automatic support:* This is a major difference between RAMASD and Gaia as the design decisions in Gaia are done by the designer in a completely ad-hoc manner while in RAMASD there is automatic support. This requires the designer to handle high design complexity.

In summary, Gaia includes a restricted role identification method and it does not support reasoning using role models impeding thus representations of goal-oriented behaviour at a high abstraction level. Furthermore, the Gaia conceptual models are not suitable for reuse in different ABS designs. In addition Gaia does not support role specialisation and it does not formally take role relationships into account. Finally, Gaia, although systematic, does not provide any support to the designers to automatically carry out a number of the design steps, whilst taking into account non-functional aspects and design heuristics. Therefore, it is concluded that Gaia involves higher design complexity than RAMASD.

8.4 Discussion

The value of RAMASD as compared to other ABS design methods with respect to design complexity has been discussed in Sections 8.2 and 8.3. This section examines RAMASD in the context of real world applications and discusses its novel aspects.

8.4.1 Real World Applicability of RAMASD

This section reports on the suitability of RAMASD for the design of ABSs for real world applications. RAMASD can generally be applied in various application domains. However, preliminary results show that RAMASD cannot scale to a satisfactory level at present.

8.4.1.1 The Generality of RAMASD

RAMASD has been successfully applied in case studies concerning supporting business systems (see Chapter 7). Furthermore, there are no restrictions in applying RAMASD in other domains as well. RAMASD could even be used to design non-agent based software but in that case roles should be defined in a different manner.

RAMASD is based on the concept of role as representation of behaviour. This is instrumental in designing ABSs supporting human activity systems, for example roles can be directly used to represent behaviours in both the business system and the ABS increasing the semantic alignment between the two systems. This makes the design requirements easier to understand and hence reduces the design complexity.

Role modelling can be used for modelling other types of systems involving autonomous behaviour as well. For example, roles have been used to describe behaviour of machines in manufacturing systems [69]. Consequently, RAMASD could be used to design ABSs in general. The role modelling technique described in Section 5.2 should be sufficient to represent all agent behaviours using appropriate roles.

RAMASD can be used for the design of traditional object oriented systems modelled using the role modelling paradigm, for example in a manner similar to the one found in the OORam object-oriented software engineering methodology [163]. Roles could be identified according to the traditional object-oriented approach and constraints on role playing by objects could be done based on the role algebra. However, the role definition should be modified in that case to reflect the non-autonomous behaviour of objects, for example that communication is done by remote method invocation and not by message passing.

8.4.1.2 The Scalability of RAMASD

In the experiments carried out in the context of the case studies RAMASD was found to not scale well for large numbers of roles and role characteristics. Due to time limitations, however, these issues were left for further research.

As far as it concerns scalability, RAMASD has currently only been tested with the simple baseline algorithm described in Section 6.5. The algorithm has worked well for simple examples involving approximately 40 roles and having on average 10 merging role constraints, 20 other role constraints and 2 general constraints. However, the algorithm becomes inefficient when the total number of roles increases, the number of merging role constraints decreases or the total number of constraints increases.

The search algorithm of RAMASD could be further improved in many respects. However, time limitations would not allow deep examination of this issue in the context of this PhD project. Therefore, this issue was left for future research (see also Section 9.4),

8.4.2 Novel Aspects of RAMASD

This section provides a critique of the philosophy and the concepts underlying RAMASD. The way that RAMASD addresses the open issues raised in Chapter 3 is justified and the merits of the main contribution, the role algebra are discussed.

8.4.2.1 The Innovative Features of RAMASD

RAMASD has a number of innovative features which make possible to explicitly take a number of design issues into account, for example design heuristics, organisational settings and non-functional aspects. These features include using performance variables to represent quantitative aspects and role models to represent various behavioural aspects qualitatively.

Extended role definition: In RAMASD, roles are representations of complex behaviour and not simple behavioural abstractions as they are elsewhere, for example in information systems engineering. The closest role definition is the one given by Kendal in [110] where roles are also considered as able to plan, have goals, and interact with other roles to achieve them. However, the role definition given by Kendal does not allow for modelling pragmatic aspects. For example, the need for a role to access some resource cannot be modelled using Kendal's role definition.

Performance variables: RAMASD offers a way to quantitatively model behavioural aspects at the role level via performance variables. In this way, pragmatic behaviour as well as abstract properties can be represented. For example, performance variables can represent required memory, as is done in Chapter 7, or they can represent non-functional issues like security levels. Modelling constructs similar to performance variables have been used in other role-based

approaches to ABSs engineering. For example, in Gaia [209] such variables are used to represent safety and liveness properties of roles. However, the variables in such approaches primarily aim to enable verification of whether the resulting agents satisfy the safety and particular properties and to the author's knowledge they are not used for allocating roles to agents. The main reason for this is that such approaches do not include any systematic methods to handle the hard search problems resulting from complex safety and liveness properties. RAMASD on the other hand, considers only constraints based on simple properties described by performance variables. The simplicity in constraint specification has the advantage that it is less hard to search for design solution satisfying the constraints. To demonstrate this RAMASD provides a simple algorithm that is guaranteed to find a solution if one exists. The author believes that this can be shown more clearly by applying efficient known search algorithms to the role allocation problem.

Specialised role models: RAMASD can represent various aspects like organisational settings and qualitative non-functional aspects by specialised role models. This approach is widely applied in the area of pattern languages. However, apart from a few exceptions, e.g. [108, 167] the synergy of different behaviours represented by role models is not explicitly considered and to the author's knowledge the role synergy has not been formalised elsewhere.

Representing special behaviours by role models reduces design complexity as the designer does not have to explicitly reason about how considering special behaviours in the system would alter the behaviour of each individual agent. For example, to enforce a number of particular organisational rules on the ABS it will be enough for the designer to request that certain role models be considered in the design solution (see also Section 5.5.3). Given that such models already exist in the role model library, the designer does not have to consider how organisational role models combine with application role models and about what roles will be allocated to each agent. Non-functional aspects can be represented by role models in a similar manner. RAMASD enforces organisational as well as non-functional requirements in a reusable manner and at a high level of abstraction. The whole approach is based on the formalisation of role relations, done by the role algebra, which is critically discussed in the next section.

8.4.2.2 The Role Algebra

The role algebra is the enabling mechanism for the various behaviours represented by roles to be combined in a systematic and rigorous manner. The primary benefit of the role algebra is that it enables designers to reason at the role level when describing behavioural constraints and have to define less constraints on role characteristics. This contributes towards both working at a high level of abstraction and automating certain steps of the design process.

Formalising the relations among roles provides a convenient way to specify design constraints at the role level. This results to less number of design constraints and hence it is more likely for a search algorithm to find a satisfactory solution. Furthermore, constraints based on role relations can increase the speed of the search algorithms when they are checked before constraints specified on agent and role characteristics.

The role algebra is instrumental for enabling work at a high level of abstraction. Based on the role algebra, RAMASD takes the proliferating view that role models should be used as first class design constructs one step further by considering that not only role models but also relations among roles can be used to describe collective behaviour. This view emphasises the fact that the overall behaviour of social entities depends both on the behaviours the entities demonstrate in particular contexts, but also on the interrelations of those behaviours (see also Section 4.2).

8.5 Summary

This chapter provided an assessment of the value of RAMASD with respect to reducing complexity in ABS design. To select an appropriate evaluation strategy, a number of methods suitable for evaluating software engineering methodologies have been considered. The result was to select a combination of descriptive and experimental evaluation as the most suitable for assessing the value of RAMASD with respect to reducing ABS design complexity.

Overall, RAMASD was shown to be superior to existing ABS design methods in several aspects. RAMASD addresses the design complexity problem by enabling designers to work at a high level of abstraction and by semi-automating the design process. Abstractability is enabled by using roles to model the agent behaviour and semi-automation is the result of applying the synthesis concept to the design process. Both abstractability and semi-automation are leveraged by a formal model of role relations, the Role Algebra. The role algebra supports high level of abstraction by enabling designers to specify design constraints at the role-model level instead of only at the role attribute level. Furthermore, it supports semi-automation of the design process by enabling automatic merging of the design synthesis problem sub-solutions, the role models, to an overall solution, the grouping of roles to agent types.

The applicability of RAMASD has been tested by applying it in two case studies. The value of RAMASD with respect to reducing design complexity was assessed using the evaluation framework of Chapter 3 and by comparing and it with Gaia in the context of the mobile workforce case study. In all cases RAMASD has been shown to be superior to other ABS methods. In particular, it has been shown that RAMASD can support design heuristics, organisational settings, collective behaviour, non-functional aspects, design process automation

and work at different abstraction levels. RAMASD is general enough to be applied to non-business oriented ABSs. It could even be able to be applied to design traditional object-oriented software (with a modified role definition). However, the current version of RAMASD is not scalable.

Chapter 9

Conclusions

This chapter revisits the hypothesis presented in Chapter 1 in the light of the work done in this project. Furthermore, it discusses limitations of the proposed approach and places the contribution of this work in the context of current ABS engineering efforts. Finally, it establishes directions for future research.

9.1 Revisiting the Research Hypothesis

This PhD project investigated the issue of reducing complexity, namely the difficulty in understanding and manipulating software artefacts, involved in the design of ABSs. In particular, it focused on the following two approaches which are known to reduce complexity in software engineering:

- Enabling ABS designers to work at a high level of abstraction; and
- Semi-automating the ABS design process

A number of additional issues are involved in implementing these two approaches, including the use of organisational settings and collective behaviour as first class design constructs, applying design heuristics and considering non-functional aspects. Therefore, the work done in this PhD aimed at developing an ABS design method which would involve less design complexity than the existing methods and that would also address the above additional issues.

To narrow the scope of the research problem so that it could be addressed in the context of a PhD project, the work was based on the view that ABS design concerns the allocation of a set of roles R , representing agent behaviours in particular contexts, to a set of agents A such that the resulting design satisfies the application requirements and any relevant design issues are taken into account. Based on this view, the problem then was how to find a method for representing agent behaviour using roles and for allocating roles to agents, which would involve less design complexity than the existing methods.

The hypothesis underlying this research was that design complexity can be reduced by formalising relevant role relations in a formal algebraic model and by developing an ABS design process based on the synthesis concept. These two approaches are in the core of RAMASD, the innovative ABS design method proposed here. In RAMASD, relations among roles, which concern allocation of roles to agents, are formally described in the role algebra and

a design process based on the synthesis concept is followed. Applying the synthesis concept provides the basis for semi-automation where the solutions to the synthesis sub-problems are manually specified by the designers or reused from a repository; and the merging of the solutions is done automatically, including the allocation of roles to agents. This allocation is made possible by the role algebra, which enables specifying design constraints at the role level. This also increases the level of abstraction involved in design activities.

The effect of these two approaches to reducing design complexity was assessed using an evaluation framework comprising a number of aspects pertinent to ABS design. RAMASD was found to support all framework aspects. Furthermore, RAMASD was compared with Gaia and was found to be superior with respect to reducing design complexity.

In summary, this thesis has demonstrated that using RAMASD reduces design complexity in ABS design and this has shown the usefulness of the two approaches underpinning the starting hypothesis.

9.2 Assessing the Thesis Contributions

The main original contribution to knowledge of this work is the overall RAMASD method, which in turn is based on a number of secondary contributions. The contributions of this work are summarised as follows:

1. *The RAMASD method.* RAMASD is new in many respects and in Chapter 8 it is shown to address the problem of complexity better than comparable existing methods. Current research trends, show an increased attention to high-level semi-automatic design of ABSs. To this end, RAMASD constitutes a fundamental step in this direction. It integrates a number of innovative aspects, including a technique for incorporating non-functional aspects and design heuristics in role models, and the synthesis-based design process enabling semi-automatic design of ABSs
2. *The role algebra.* In the domain of ABS engineering, there is no formal model to describe relations among roles concerning assignment of roles to agents. Formal models involving roles can be found in other domains, such as Role-based Access Control, for instance [14], and in the areas of role-based pattern languages, for instance [54]. but their complexity renders them unsuitable for reducing complexity in designing ABSs. In contrast, the role algebra was developed with the aim of being practical and easy to understand. This contribution is important because it enables reasoning at a high level of abstraction and semi-automation of the design process while hiding from the designer the details of role attributes. In order to keep the model practical it was attempted to

include only a small number of basic relations. However, the model is open-ended, and more role relations can be added as needed.

3. *Classification scheme and evaluation framework.* Chapter 2 and Chapter 3 propose a classification and comprehensive evaluation of current ABS engineering approaches focused on design complexity. To the best of the author's knowledge, no similar evaluation framework for ABS engineering methodologies currently exists. This contribution is important because it offers a systematic way to assess ABS engineering methodologies and it can be easily extended to cover other aspects of ABS engineering and not only those concerning design complexity

9.3 Limitations of RAMASD

RAMASD was the outcome of a PhD project which had to be completed in three years. Therefore, it is based on a number of assumptions the validity of which would require more time to explore. RAMASD limitations include static role allocation, primitive models of non-functional aspects and inefficient search algorithms.

It has been argued in this thesis that it is preferable to design an ABS once and for all on design time. However, there are cases where the behaviour of agents may need to change dynamically on run-time, for example when agents should be able to roam the internet and interact in unknown domains. The current version of RAMASD does not explicitly consider such possibilities. This problem can be partially overcome by using role modelling workarounds. For example, some generic intermediary role, as the one used in the example described in Section 5.5.3, can be used to represent all alternative interactions of an agent with unknown hosts. The guest agent will be concerned only with interacting with the intermediary role which will be played by a host agent. The intermediary role will then interact with other roles in the particular host environment based on host specific rules and information. This workaround is particularly suitable for enforcing organisational rules but generally it has many disadvantages, for example all the dynamic changes to agent behaviour need to be exposed to the host environment. Furthermore, a large number of behavioural alternatives need to be encapsulated in the intermediary role. Hence, this workaround may result to increased design complexity.

Another limitation of RAMASD is the way that it models non-functional aspects. It currently uses the simplistic assumption that the relationship between the agent's performance variables and those of its roles is linear. This was illustrated in the mobile workforce case study (Chapter 7) where the memory of an agent was assumed to equal the sum of the memory required by each role played by the agent. In general however, this may not be the case, and more general models for representing these relations are necessary.

9.4 Further Work

The research carried out in this PhD project identified a number of areas where it would be interesting to undertake further work. These areas include extending the role algebra to allow dynamic role allocation, applying RAMASD in other areas, for instance agent-based web services, and developing efficient role allocation algorithms. In addition, interesting further research topics include developing powerful models of non-functional aspects based on role characteristics and developing a framework for classifying and assessing organisational patterns.

- *Extending the Role Algebra to support dynamic role allocation.* Addressing the current limitations of RAMASD includes the interesting problem of extending the role algebra to support both dynamic and static role allocation. Such a model would be particularly useful in conceptualising the adaptive behaviours that agents are required to exhibit in contemporary dynamic environments. The author has already started work in this direction both individually and in the context of the Agentcities Workgroup concerning Engineering Self-Organising Applications [178]. The first results of the author's individual efforts in this direction are expected to be ready for publication in the first half of 2003.
- *Applications of RAMASD in other areas.* RAMASD is particularly suitable for designing ABSs targeting specific application domains. For example, formalisation of role relations can be used for automatic creation of agent-based business services. Agreements between business parties concern not only products or services offered but also qualitative aspects such as service quality and time constraints. Such agreements are termed *Service-Level Agreements (SLAs)*. Taking service-level agreements into consideration whilst designing supporting software is a hard problem [125]. In RAMASD, different services can be represented by appropriate roles and service combination constraints can be described by an extended version of the role algebra. In this way, only valid service bundles will be created and assigned to agent components tasked with overseeing the service provision processes. Dynamic formation of services could then be supported using a version of RAMASD that allows dynamic role allocation as discussed above.
- *Improved search algorithms.* Interesting work could be done in the direction of finding faster search algorithms to be used for allocating roles to agents, for example in the areas of constraint satisfaction problem solving and heuristic search. Different types of constraints and their properties, for example the looseness and density properties described in [189], can impact the efficiency of the role allocation algorithm. Similar work is also done in the area of databases and the author believes that algorithms used for database query satisfaction could be reused for efficient role allocation to a certain extent. Finally, useful

search algorithms could be adopted from the area of automatic software engineering and in particular from algebraic programming.

- *Efficient models linking agent and role characteristics.* As mentioned in the previous section, RAMASD currently considers only simple and intuitive models linking agent and role characteristics. Addressing this problem can be pursued along a number of directions, for example each role can be associated with a rule-base making it '*intelligent*'. This follows ideas from automated software design [124] where rulebases are used for specification of software components. In such an approach, agent types would also be associated with rulebases. In order for roles to be allocated to a particular agent type, the resulting rulebases should be consistent, namely any constraints would need to be satisfied. Such an approach would also give rise to interesting research for appropriate search algorithms as mentioned above.
- *Organisational patterns.* An interesting direction for further work concerns organisational patterns. Organisational patterns are quite powerful in specifying the overall behaviour of an ABS but currently there is no systematic way for the designers to select which organisational patterns to use. Hence, an appropriate organisational pattern classification and assessment framework is required. This view is along the lines of similar views expressed by other authors. For example Wooldridge in [212] stresses the need of establishing quantitative criteria for evaluating organisational patterns and Coplien in [39] advocates a qualitative organisational pattern classification. In particular, Coplien identifies recurring patterns of interaction in business organisations and attempts to discover recurring matches between those patterns and some qualitative measure of "goodness". It would be quite interesting to attempt to combine the two approaches in a comprehensive framework where additionally relations among pattern roles would be specified using the role algebra. In this way, designers would have a systematic way of both selecting suitable organisational patterns and combining them with application functionality to design ABSs.

9.5 Concluding Remarks

This PhD project has resulted in important findings regarding complexity in ABS design. The findings include identifying issues relevant to design complexity and ways in which these issues should be addressed by ABS design methods. However, the most important result produced out of this PhD work is the overall approach which makes it possible to reduce the level of specification detail and delegate part of the design work to an automatic tool.

Finally, ABS engineering is a research area where elements of multiple research areas can be fruitfully combined to facilitate the engineering task. It is the author's belief that combining

concepts from different areas to accurately represent the agent behaviour and attempting to automate the engineering process is the correct approach for effectively engineering real world ABSs.

APPENDICES

Appendix A Evaluation of ABS design approaches

This Appendix reviews the ABS engineering methodologies discussed in Chapter 2 in detail and presents the results of their evaluation as far as it concerns the design of ABSs. In particular, a representative approach for each class in the classification scheme proposed in Section 2.4 is reviewed and assessed as far as it concerns its support for ABS design based on the evaluation framework introduced in Section 3.1. A comparative evaluation of the assessed ABS engineering approaches is presented in Section 3.2.

A.1 RAPPID

RAPPID (Responsible Agents for Product-Process Integrated Design) is a domain specific ABS engineering approach that targets the domain of collaborative product design [158]. ABSs developed with RAPPID aim to assist human product designers manage product characteristics across different functions and stages in the product design life cycle [155].

A.1.1 Overview of RAPPID

In RAPPID, each human with a stake in the design (including designers, manufacturing engineers, and marketing and support staff), each component of the design itself and the characteristic of each component is represented by an agent. Agents representing humans are called *Component Agents* while other agents are called *Characteristic Agents*. Component Agents and Characteristic Agents trade with one another for design constraints, requirements, and manufacturing alternatives, and the resulting ABS provides a mechanism that yields product designs faster than conventional techniques. RAPPID agents are active software objects with varying degrees of intelligence.

Figure A.1 shows a product design decomposed into Component Agents (rounded rectangles), each with one Characteristic Agent (ovals) for each dimension in the design space. For example, the "*SS.Weight*" Characteristic Agent might represent the constraint that the entire product must weight between 5 and 10 kg. The topmost Component Agent represents the complete product and is the concern of the Chief Engineer, who reflects the Customer's requirements in the initial allocation of design space. The bottommost Component Agents are either custom-manufactured or selected from an on-line Parts Catalogue. Designers, who typically have responsibility for

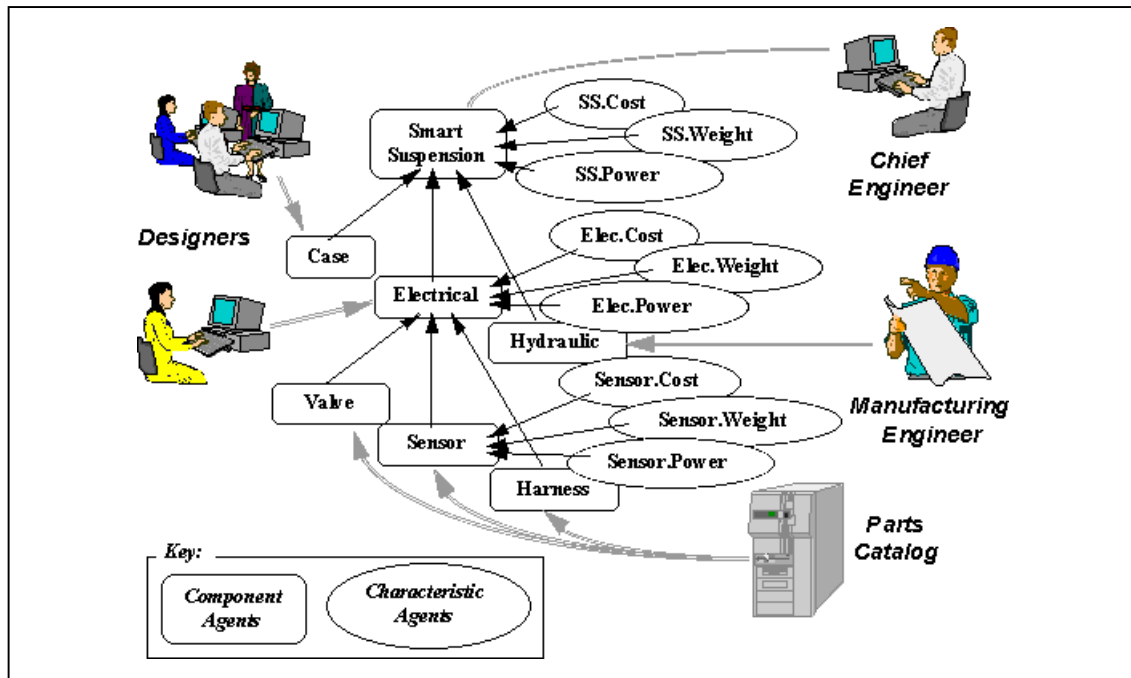


Figure A.1: The RAPPID ABS architecture

intermediate levels of the product tree, propagate the constraints from the top and bottom of the tree toward each other. Each Component Agent (either automatically or under guidance from its Designer) buys and sells design space allocations to and from other Component Agents.

RAPPID has been used in many application areas including unmanned air vehicles design [154] and container ship design [152].

A.1.2 Evaluation of RAPPID

RAPPID is limited in the sense that it targets a specific application domain. Therefore, the produced agents are restricted as far as it concerns their possible uses. Furthermore, RAPPID does not clearly support the design phase of the software engineering life cycle. Instead, it only provides a generic approach and guidelines about how the agents and the ABS should be built. The exact system functionality is supposed to be decided by the ABS engineer in an ad-hoc manner. Furthermore, a RAPPID ABS is viewed as a hierarchy of problem solvers and the system conceptualisation progresses from the upper levels of this hierarchy in a top-down manner.

Concepts	Concept definition	Design in scope	Heuristics support
	\supseteq	–	–
Models	Organisational Settings	Collective behaviour	Non-Functional aspects
	–	–	–
Process	Design Perspective	Support for reuse	Design automation
	↓	–	–
Pragmatics	Generality	Abstractability	Tool support
	○	–	–

Table A.1: Evaluation of RAPPID

RAPPID does not include modelling mechanisms to explicitly represent organisational settings, collective behaviour and non-functional aspects. In addition, there is no systematic way to formally support design heuristics, to reuse design knowledge and to automate the design process. Finally, the generality of the RAPPID approach is low as it aims at creating systems to support the specific application domain of product design. Furthermore, there is no formal support for reasoning at different levels of abstraction. Finally, there is no support for RAPPID by a software tool due to its ad-hoc and informal nature (The only tools associated with RAPPID are some spreadsheets and Java-based editors which simply facilitate editing of ABS descriptions [158]). The evaluation of RAPPID is summarised in Table A.1.

A.1.3 Strengths and Weaknesses of RAPPID

Although this approach has proven useful in developing research prototypes demonstrating conceptual models of an application domain, it is not applicable to large and complex agent systems targeting real world applications. The main disadvantage is that the generic guidelines for designing the ABS that RAPPID provides make it difficult for the ABS designer to consider the overall picture where the guidelines may not be completely applicable or where there may be conflicting requirements. This may result to design errors. Another major problem is that RAPPID is limited to the domain of product design. As far as it concerns design effort, RAPPID has the disadvantage that there is no systematic way for justifying and reusing design knowledge. This inconvenience for the designer increases as there is no support by a software tool.

The only advantage of RAPPID is that the approach is not difficult to use, as it does not involve complicated models and a large number of concepts. However, this advantage deteriorates due to the lack of systematic methods and formality as the size of the designed ABS increases.

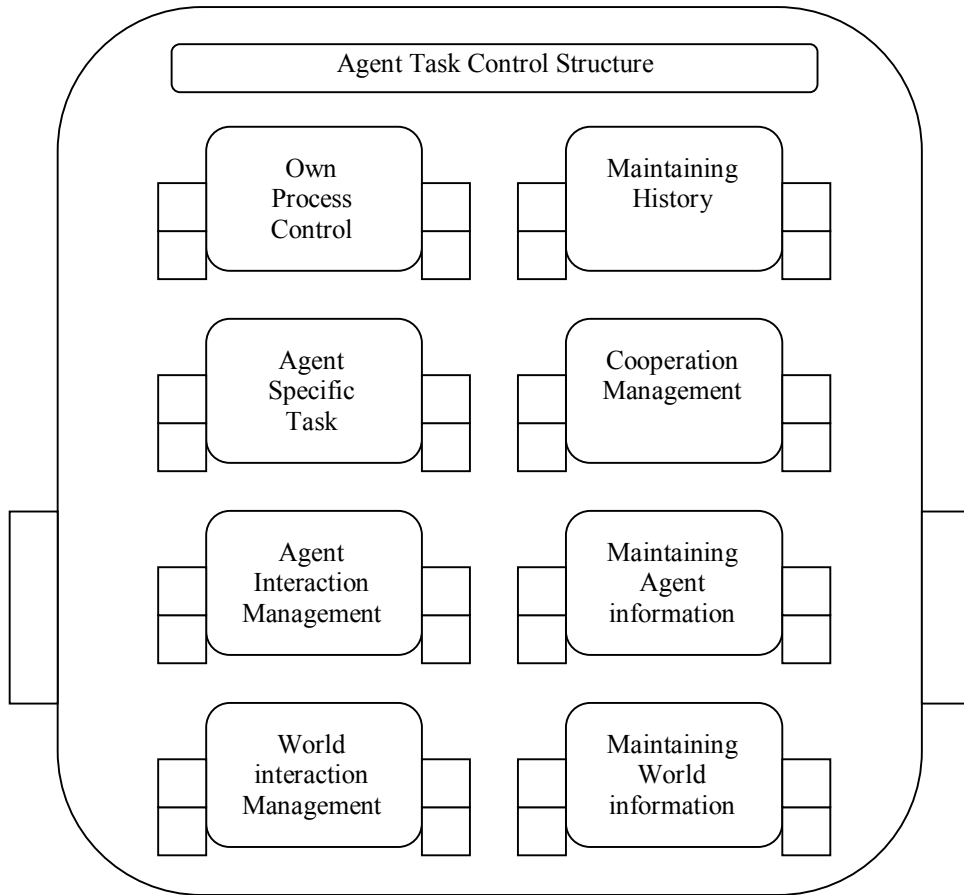


Figure A.2: A generic agent model in DESIRE

A.2 DESIRE

DESIRE is an ABS-modelling framework, which can be used for conceptual specification, behavioural simulation and prototype generation of ABSs [21, 22].

A.2.1 Overview of DESIRE

The DESIRE approach adopts a compositional view of agents and ABSs. According to this view, the entire functionality of the system is modelled as a series of interacting, task-based, and hierarchically structured components. Each task can be either *primitive* or *composite*. A task hierarchy is constructed by applying a recursive top-down decomposition process on the initial system task. The compositional view of DESIRE considers each individual agent and the whole ABS as a collection of components that represent task solving units. The dynamic patterns of interactions in the ABS are modelled as interactions among tasks of the same or different agents at different levels of reasoning.

The DESIRE framework proposes two models that should be specified by the ABS designer. The *intra-agent* model contains the expertise descriptions of domain tasks, the knowledge requirements and the reasoning capabilities for solving these tasks. The *inter-agent* model,

describes the expertise to perform and guide coordination, cooperation and social interaction among agents.

In DESIRE, the agent architecture is defined by composing primitive component models that are directly related to agent tasks. Existing generic agent models can be used to design a specific agent model. During the design process, relevant components in a generic model are refined by (1) more detailed analysis of the tasks of which such components are comprised and/or (2) inclusion of specific domain knowledge.

Within the DESIRE framework, knowledge is represented at three different levels: *conceptual level*, *detailed level* and *operational level*. The representation at the operational level is automatically generated from the representation at the detailed level. Furthermore, when the specified ABSs are small and agents have simple architectures, it is possible to simulate the ABS behaviour and systematically experiment for a number of parameters concerning the agent environment. In addition, DESIRE has the advantage that the specifications and their semantics can be formally described using temporal logic as a base. This enables proving various properties about the system during the verification and validation phases of the software lifecycle.

An example of the representation of the compositional structure of a generic DESIRE agent is depicted in Figure A.2. In this model, eight agent tasks are performed by the eight internal agent components: control of an agent's own processes (Own Process Control), interaction with other agents (Agent Interaction Management), maintaining knowledge of other agents' characteristics (Maintain Agent Information), interaction with the external world (World Interaction Management), maintaining knowledge of the external world (Maintain World Information), maintaining information regarding past observations and interactions (Maintain History), managing cooperativeness (Cooperation Management), and performance of agent specific tasks (Agent Specific Tasks).

DESIRE is based on a formal specification language and it is associated with a variety of tools aiming to assist the designer in creating, verifying and simulating the ABS. Those include a graphical editor, a specification compiler that generates Prolog code and various debugging and monitoring tools. Furthermore, DESIRE can be applied in various application domains. In particular, DESIRE has been used by a number of companies and research institutes (such as chemical industry, financial sector, software industry, institutes for environmental studies) to develop operational systems for a number of complex tasks (including systems for diagnosis, design, routing, scheduling and planning).

Concepts	Concept definition	Design in scope	Heuristics support
	\diamond	\sqrt	–
Models	Organisational settings	Collective behaviour	Non-functional aspects
	–	–	–
Process	Design perspective	Support for reuse	Design automation
	\downarrow	\sqrt	–
Pragmatics	Generality	Abstractability	Tool support
	\otimes	\sqrt	\sqrt

Table A.2: Evaluation of DESIRE

A.2.2 Evaluation of DESIRE

The DESIRE approach covers the analysis and the design phase of the software development life cycle. However, DESIRE assumes a specific, task based agent architecture and therefore it is restricted in this sense. Furthermore, the DESIRE approach is applied in a top-down fashion.

Major disadvantages of DESIRE are that it does not support explicit modelling of organisational settings, collective behaviour and non-functional aspects. It is the responsibility of the agent-system designer to incorporate those aspects implicitly by task-based modelling. As DESIRE is based on a generic decomposition framework, parts of design knowledge can be reused. For example, as described in [20] some tasks that are generic in some domain can be specialised as required. Furthermore, the formality inherent in the DESIRE approach makes possible realising DESIRE in a software tool. However, the design process cannot be automated to any extend as the agent components should be known before the specification of the agent behaviour is made. Furthermore, DESIRE does not provide any systematic and formal support for applying design heuristics. Although some heuristic design rules could be modelled using the DESIRE specification language this needs to be done intuitively by the designer. Finally, DESIRE formally supports specifying interactions among task components at different levels of abstraction, which reduces design complexity. A summary of the evaluation of DESIRE is given in Table A.2.

A.2.3 Strengths and Weaknesses of DESIRE

The major advantage of DESIRE is the high degree of formality, which enables verification and consistency checking of specifications of ABSs. Furthermore, an additional advantage is that it provides support for reuse, which significantly reduces development effort [141].

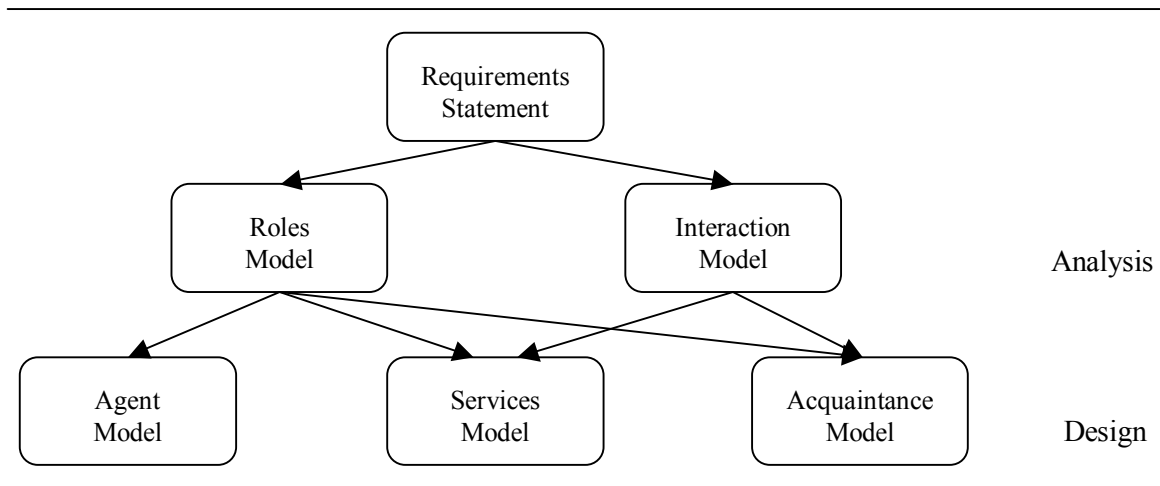


Figure A.3: Relations between Gaia models

However, the commitment of DESIRE to a specific agent architecture impedes its general applicability to a broader range of problems. Furthermore, since the agent components should need to be decided before the approach is applied, the design of the ABS cannot be automated. This makes difficult to apply DESIRE to design large ABSs.

A.3 Gaia

An example of an approach combining agent theoretic concepts with object oriented software engineering principles is Gaia [209]. Gaia is a general methodology supporting the design of both the individual agent architecture as well as the agent organization.

A.3.1 Overview of Gaia

Gaia [209] is one of the first systematic methodologies to view an ABS as an organisation of agents. The Gaia methodology includes two analysis models and three design models, as outlined in Figure A.3.

The first step in the Gaia *analysis* process is to find the *roles* in the system, and the second is to model *interactions* between the roles found. Roles in Gaia consist of four attributes: responsibilities, permissions, activities and protocols. *Responsibilities* are of two types: *liveness properties* indicating that the role has capabilities that add something good to the system, and *safety properties* that prevent and disallow something bad to happen to the system. *Permissions* represent what the role is allowed to do, in particular, which information it is allowed to access. *Activities* are tasks that a role performs without interacting with other roles. *Protocols* are the specific patterns of interaction, for example, a seller role can support different auction protocols such as “English auction”. Gaia has formal operators and templates for representing roles and

their belonging attributes and it has schemas that can be used for the representation of interactions.

Concepts	Concept definition	Design in scope	Heuristics support
	\succ	$\sqrt{}$	–
Models	Organisational settings	Collective behaviour	Non-functional aspects
	–	–	–
Process	Design perspective	Support for reuse	Design automation
	\downarrow	–	–
Pragmatics	Generality	Abstractability	Tool support
	\emptyset	–	–

Table A.3: Evaluation of Gaia

In the Gaia *design* process, the first step is to map roles into *agent types*, and then to create the right number of *agent instances* of each type. The second step is to determine the *services model* needed to fulfil a role in one or several agents, and the final step is to create the *acquaintance model* for the representation of communication between the agents.

A.3.2 Evaluation of Gaia

Gaia considers only cooperative agents that act towards a common goal whilst many agent systems have to deal with resolving a number of conflicts between agents. In addition, the problem domain is assumed to not contain any conflict situations that would need to be resolved. Furthermore, although there is no restriction regarding the internal agent architecture and the programming language in which the agents will be implemented, there is a restriction of size as Gaia targets small ABSs of about 100 agents. Agents are further assumed to not be mobile.

The Gaia approach covers the analysis and design phase of the software development process on a very high level. The resulting ABS designs are intended as input to traditional software engineering methods that refine the high level designs into particular implementations. Gaia does not support explicit modelling of organisational settings, of collective behaviour and of non-functional aspects. Furthermore, since the approach is completely informal the design process cannot be automated to any extend. In addition, there is no tool support for GAIA.

Gaia does not support reuse since it does not provide any guidelines or techniques about how existing specification models can be reused in the design of ABSs. Finally, there is no formal

support for work at different levels of abstraction and design complexity is not handled in Gaia. The evaluation of the Gaia approach is summarised in Table A.3.

A.3.3 Strengths and Weaknesses of Gaia

An advantage of Gaia is that it provides a concrete set of models that capture almost most relevant aspects of the target ABS. Furthermore, the application of Gaia is quite straightforward even for inexperienced users and it is not committed to any particular agent architecture such as BDI or similar.

A major weakness of Gaia is that it is simply defined on top of other object-oriented software engineering techniques. Therefore, it may be difficult to produce sufficient implementations from the Gaia design models. A way to mitigate this problem could be to include an environment model in the set of GAIA models, for example as is done in MAS-CommonKADS [91], but still a systematic way to refine high-level designs to implementations would be required. Additional considerable weaknesses of Gaia is the lack of explicitly modelling non-functional aspects and supporting the use of organisational settings and collective behaviour as first class design constructs and the lack of automation of the design process.

A.4 Tropos

Tropos [23, 31, 77] is an approach to ABS engineering, which originated from the area of information systems engineering.

A.4.1 Overview of Tropos

Tropos is based on two key concepts: the *notion of agent* and the concept of *mentalistic attitudes*, for example beliefs, capabilities, actions and plans that characterize an agent. These ideas are used in all phases of software development. Unlike other ABS engineering approaches, Tropos also covers the very early phases of requirements analysis and thus provides the software designer with a deeper understanding of the environment in which the ABS will operate. Tropos targets the realization of ABSs that automate processes normally carried out by groups of humans within business organisations. Tropos supports five phases of software development:

- *Early requirements.* This phase is concerned with the understanding of a problem by studying existing business organisational settings. The output of this phase is an organisational model, which includes relevant actors and their respective dependencies. Actors are characterized as having goals, which they would be unable to achieve in isolation.
- *Late requirements.* The system under realization is described within its operational environment, along with its relevant functions and qualities. This description models the

system as a number of actors, which have social dependencies with other actors in their environment.

Concepts	Concept definition	Design in scope	Heuristics support
	\diamond	$\sqrt{}$	–
Models	Organisational Settings	Collective behaviour	Non-Functional aspects
	–	–	–
Process	Design perspective	Support for reuse	Design automation
	\downarrow	–	–
Pragmatics	Generality	Abstractability	Tool support
	\oplus	–	–

Table A.4: Evaluation of Tropos

- *Architectural design.* The system's global architecture is defined in term of subsystems, interconnected by data and control flows. Subsystems are represented as actors, while data and control interconnections correspond to actor dependencies. Actor capabilities and agent types (agents are special kinds of actors) are specified. This phase finishes with the specification of agents within the system.
- *Detailed design.* Each agent of the system is defined in detail, in terms of internal and external events, plans and beliefs, and agent's communication protocols.
- *Implementation.* The actual implementation of the system is carried out, consistently with the detailed design.

A.4.2 Evaluation of Tropos

Tropos is tailored to software systems that will operate in a business organisational context. Therefore, it makes possible to use the same concepts to describe the organisational environment within which the ABS will eventually operate, as well as the system itself. Apart from that, the only other restrictive premises of Tropos are that it focuses on the BDI architecture.

Furthermore, Tropos provides a smooth transition from analysis to design and implementation since all phases are agent-oriented. This minimises the need to explicitly transform agent concepts to traditional object oriented concepts and constructs, for example classes and methods, in order to implement them, as is the case in MASE [186]. Tropos follows the JACK [135] agent-building toolkit as far as it concerns agent architecture and implementation concepts. Therefore, Tropos models can be implemented in a rather straightforward manner.

In addition, Tropos is to our knowledge the only static approach that provides support for non-functional aspects. Non-functional aspects are represented by appropriate *softgoals* based on the *i** framework [35]. The representation is done in a qualitative manner depending on the designer to introduce sub-actors that would contribute positively to the fulfilment of softgoals.

However, Tropos does not support neither organisational settings nor collective behaviour as first class design constructs although it is tailored to supporting business organisations. Another weakness of Tropos is that it does not provide formal and systematic support for applying design heuristics. In addition, Tropos does not support reuse since all design decisions have to be done every time from the beginning. Furthermore, the design complexity is not addressed in Tropos, as there is no formal support for the designer to work at different levels of abstraction. Finally, to our knowledge there is currently no tool support for Tropos.

The evaluation of Tropos is summarised in Table A.4.

A.4.3 Strengths and Weaknesses of Tropos

Tropos is a comprehensive methodology, which offers additional advantages compared with other approaches, for example support for non-functional aspects, and it has a wide scope covering all phases of software engineering lifecycle. A significant advantage of Tropos is that it successfully integrates aspects from known methodologies for requirements gathering with aspects from information systems engineering and agent concepts to create a comprehensive ABS engineering approach. As a result, Tropos is far more powerful regarding requirements gathering and analysis than other ABS engineering approaches.

However, Tropos provides poor support for ABS design. In particular, it does not support organisational settings and collective behaviour as first class design constructs. Furthermore, it does not provide any support for design heuristics, for reuse of design knowledge and for design process automation. Finally, Tropos does not handle design complexity, as it does not support the designers to work at different levels of abstraction.

A.5 MESSAGE

MESSAGE/UML [30, 60] (Methodology for Engineering Systems of Software Agents) is an AOSE methodology, which builds upon current software engineering best practices covering analysis and design of MAS. It has well defined concepts and a notation that is based on UML.

A.5.1 Overview of MESSAGE/UML

MESSAGE/UML is primarily focusing on considering a wide range of agent concepts in the conceptual modelling of ABSs. The contributions of MESSAGE also include diagrams for viewing these concepts, which are based on extensions of the UML modelling language.

message is an object communicated between agents carrying information. The various relations between knowledge concepts of MESSAGE/UML can be seen at Figure A.4.

MESSAGE/UML puts the emphasis on the analysis phase. Similar to many other software engineering approaches, MESSAGE examines the existing system using a number of models based on different views of the system. The proposed MESSAGE/UML analysis models are represented by diagrams, which are extensions of UML class and activity diagrams. Currently, six analysis views are considered: organisation, goal/task, agent/role, delegation, workflow, interaction and domain.

The *Organisation view* considers the various coarse-grained relationships, for example aggregation, power, and acquaintance relationships, between various entities in the system, including agents, organisations, roles and resources. The *Goal/Task view* describes goals, and tasks and the dependencies among them. Goals and tasks can be linked by logical dependencies to form graphs showing, for example, the decomposition of high-level goals into sub-goals, and how tasks can be performed to achieve goals. Graphs showing temporal dependencies can also be drawn based on UML activity diagrams. The *Agent/Role view* focuses on the individual agents and roles. For each agent/role it uses schemata supported by diagrams to describe its characteristics, for example what goals the agent/role is responsible for and what resources it controls. The *Interaction view* describes the characteristics of interactions, for example the initiator, the collaborators and the relevant information supplied/achieved by each participant of the interaction. Larger chains of interaction across the system, for example corresponding to use cases, can also be considered in this view. Finally, the *Domain view* describes the domain specific concepts and relations that are relevant for the system under development, for example for a system dealing with making travel arrangements relevant concepts would include trip, flight and ticket.

The analysis models are produced by stepwise refinement and can be created at different levels of abstraction. The top level of decomposition is referred to as level 0. This initial level is concerned with defining the system to be developed with respect to its stakeholders and environment. The system is viewed as a set of organisations that interact with resources, actors, or other organisations. Actors may be human users or other existing agents. Subsequent stages of refinement result in the creation of more detailed models numbered in the same way, for example level 1. In level 1, the structure and the behaviour of entities such as organisation, agents, tasks, goals domain entities are defined. In the current MESSAGE/UML project, only level 0 and level 1 have been considered.

Concepts	Concept definition	Design in scope	Heuristics support
	><	–	–
Models	Organisational settings	Collective behaviour	Non-Functional aspects
	–	–	–
Process	Design perspective	Support for reuse	Design automation
	↕	–	–
Pragmatics	Generality	Abstractability	Tool support
	⊗	–	√

Table A.5: Evaluation of MESSAGE/UML

The refinement of level 0 analysis models can be done following three possible approaches: *organisation-centred*, *agent-centred* and *goal/task-centred*. In *organisation-centred* approaches the focus is on analysing the system overall properties, for example the system structure and the services offered. Subsequently, the agents that are able satisfy those properties are identified during the refinement process. *Agent-centred* approaches mainly focus on the identification of agents needed for providing the system functionality and subsequently they determine the most suitable organisation according to system requirements. Finally, *Goal/task-centred* approaches are based on functional decomposition. System roles, goals and tasks are systematically analysed to determine the most appropriate problem-solving methods, and decomposition and exception handling mechanisms required. Therefore, considering the overall structure of goals and tasks in the goal/task view the most appropriate agents and organisation structure for achieving those goals/tasks can be determined.

A.5.2 Evaluation of MESSAGE

MESSAGE is intended to be applicable to a variety of agent cognitive architectures and there are no restrictive assumptions regarding the domains it can be applied in. Furthermore, although there are some heuristic rules as to how the various MESSAGE/UML models can be created [187] in general MESSAGE/UML does not provide systematic support for heuristics regarding software design.

MESSAGE/UML explicitly models organisational settings. Those models of organisational settings can be directly adopted when model refinement is done in a top-down fashion and can be determined from goal/task models when refinement is done bottom-up. This is not the case for non-functional aspects, however, as they are not explicitly modelled. MESSAGE/UML furthermore explicitly models interactions and collective behaviour.

Due to its completely informal nature, MESSAGE/UML cannot be automated and in fact there is no systematic decision-making support in ABS design. The MESSAGE/UML approach is implemented in the MetaEdit software tool [29].

The MESSAGE/UML approach is general and it can be applied in any application domain. In addition, the approach is extensible in the sense that additional levels of detail can be defined for analysing specific aspects of the system dealing with functional requirements and non functional requirements such as performance, distribution, fault tolerance, security. Such extensions are expected to be accompanied with appropriate techniques for consistency checking between subsequent levels. However, although support for non-functional aspects is in this sense possible, to our knowledge there are not examples of such extensions in the literature. Finally, the abstractability of message results in low complexity.

The evaluation of MESSAGE is summarised in Table A.5.

A.5.3 Strengths and Weaknesses of MESSAGE/UML

The major advantage of MESSAGE/UML is the comprehensive coverage of different facets of the system during the analysis phase. An additional advantage is the flexibility it provides to the designer allowing her to work in both a top-down and a bottom-up fashion depending on the application requirements. Two important additional advantages are the applicability of MESSAGE/UML to all application domains and to all agent types are additional advantages.

The major disadvantage of MESSAGE/UML is that it does not currently provide any support for the design phase of the agent-base system. This requires the designer to handle the design in an ad-hoc manner after creating the analysis models. The MESSAGE/UML approach is the subject of on-going research [60] and therefore this lack of design support may be addressed in the near future.

A.6 Zeus

The Zeus agent development approach is closely related to the Zeus agent development toolkit [147]. This relation enables increased tool support and rapid development of ABSs.

A.6.1 Overview of Zeus Agent Development Methodology

In common with most other structured development methodologies, the Zeus ABS engineering approach consists of *analysis*, *design* and *realisation* activities, as well as *runtime support* facilities that enable the developer to debug and analyse their implementations (Figure A.5).

The purpose of the initial *analysis* phase is to model and understand the application problem.

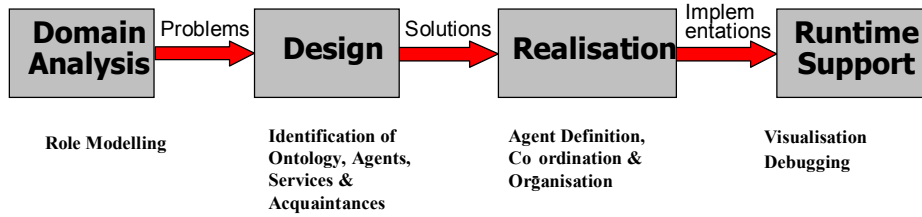


Figure A.5: The Zeus agent development methodology

The Zeus methodology does not explicitly prescribe any particular approach to problem analysis. Instead, it allows developers to “*mix and match*” their own favourite approaches, for example *use cases* [36]. However, the recommended technique is *role modelling* [38].

The *design* phase involves linking role responsibilities to agent characteristics, and deciding on the roles allocated to each agent. Role responsibilities correspond to *tasks* that an agent is capable of carrying out and domain knowledge is described with *facts* store in the internal knowledge base of agents. Furthermore, the proactive behaviour is represented with *goals* agents try to achieve. While the analysis process involved understanding the problem requirements, the design process involves expertise, knowing when and how to reuse and adapt existing proven solutions.

The objective of the *agent realisation* phase is to realise working agent implementations from the conceptual designs created during the design stage. The agent realisation process consists of several steps, which are closely coupled to the levels of abstraction that exist within a Zeus agent. Zeus agents have the architecture depicted in Figure A.6. It consists of the *Definition layer* that implements the reasoning and learning capabilities of the agents, the *Organization layer* that manages and maintains the relationships with other agents and the *Coordination layer* that is responsible for the coordination among agents and contains the necessary negotiation knowledge. Furthermore, the *Communication layer* provides the communication facilities for the communication among agents and the *API layer* that serves as the programmatic interface between Zeus agents and traditional Java objects.

The Zeus toolkit provides an extensive set of editors allowing the designer to easily specify different types of agents and the characteristics of the ABS including the organisational relations and the interactions among agents, for example coordination and negotiation models. The designer has the option to either use predefined interaction models or create new ones as required. Furthermore, the ABS designs can be transformed to Java source code within the *Agent Generator* tool.

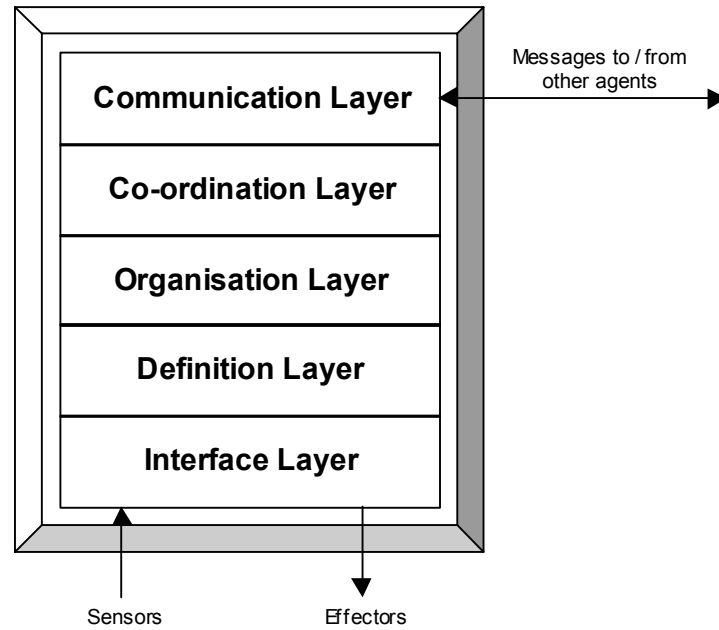


Figure A.6: The Zeus agent architecture

The Zeus approach also provides support for testing, debugging and optimising the generated ABSs. This consists of a suite of *runtime support* tools that are available in the form of the *Visualiser* agent. The Visualiser agent is constantly executing through the lifecycle of the ABS gathering statistics regarding the ABS performance.

Additional debugging and testing tools include the *Society Tool*, which monitors the messages exchanged among agents, the *Report Tool*, which displays the current state of agent task decomposition and execution, the *Micro Tool* that is used to inspect the internal state of an agent, the *Control Tool*, which can remotely modify the internal state of an agent, the *Statistics Tool*, which generates statistics regarding the performance of the ABS and the *Video Tool*, which can record and replay ABS lifecycle executions.

Further details about the Zeus agent building environment are given in Chapter 6 where the way that Zeus was extended to support RAMASD is discussed.

A.6.2 Evaluation of Zeus Agent Development Methodology

The Zeus Agent Development methodology (ZAD) is closely related to the Zeus agent building toolkit. Therefore, it involves restrictive premises. For example, all must conform to the agent architecture of Figure A.6.

The Zeus documentation suggests a number of heuristics that could be followed during ABS design, for example the *sphere of responsibility* and the *point of interaction* [38].

Concepts	Concept definition	Design in scope	Heuristics support
	\diamond	\sqrt	–
Models	Organisational Settings	Collective behaviour	Non-Functional aspects
	\sqrt	\sqrt	–
Process	Design perspective	Support for reuse	Design automation
	\uparrow	\sqrt	–
Pragmatics	Generality	Complexity handling	Tool Support
	\emptyset	–	\sqrt

Table A.6: Evaluation of the Zeus agent development methodology

However, there is no formal support for applying those heuristics in the design of the ABS. The designer is responsible for proceeding with the design based solely on intuition and experience.

ZAD considers explicit models of organisational settings and interactions. This is done based on role modelling. Furthermore, ZAD covers most phases of the software engineering life cycle. Design is done in a bottom-up fashion as the agent components are composed from primitive behaviours described by role models.

As far as it concerns reuse, the toolkit environment provides the capability of storing and retrieving previous design decisions; therefore, reuse of design knowledge is possible. This is not the case with automation, however, as the design decisions have to be taken by the designer alone without any support from the toolkit. In addition, there is no formal support for working at different levels of abstraction and hence the designer is not adequately assisted in handling design complexity.

Zeus is a general approach, which can be applied in many environments. However, there are some limits imposed by the toolkit environment and the implementation technology, for example, Zeus agents cannot be mobile. Apart from that, the toolkit provides substantial support to the designers. The evaluation of Zeus agent development methodology is summarised in Table A.6.

A.6.3 Strengths and Weaknesses of Zeus

All in all, the freely available Zeus toolkit provides a comfortable and flexible framework for the development of ABSs. However, Zeus is limited to particular agent architecture and capabilities that may not be suited for all kinds of applications domains, for example where agent mobility is required. In addition, a major weakness of Zeus is that the design process is completely informal and therefore it cannot be automated providing extra assistance to the

designers. Finally, the lack of formality in Zeus impedes appropriate handling of design complexity by working at different levels of abstraction.

A.7 KARMA/TEAMCORE

Whilst all methods reviewed up until now are static, where design is done once before the system execution, a number of approaches use the opposite paradigm where agent systems are self-organised dynamically at run-time, KARMA/TEAMCORE [192, 193] is such an approach. KARMA/TEAMCORE enables rapid integration of existing agents to agent organisations based on the application requirements and therefore it reduces the development effort. Agent organisations can be modified on run-time considering dynamic changes in application requirements and the agent environment.

A.7.1 Overview of KARMA/TEAMCORE

The KARMA/TEAMCORE philosophy is that instead of engineering ABSs from scratch, it would be better to search and recruit appropriate agents that already exist in the cyberspace. The idea is that the search for appropriate agents should be done automatically on

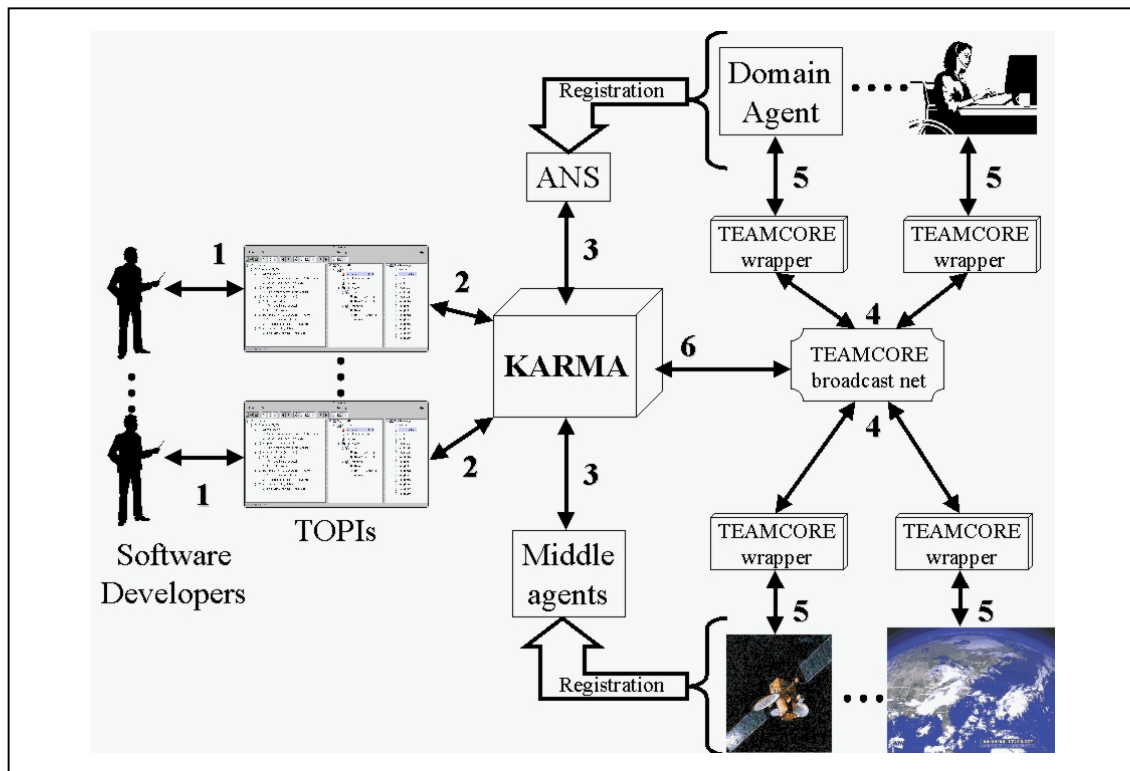


Figure A.7: The KARMA/TEAMCORE Framework

run-time, enabling thus reorganisation of the ABS when required. In this way, even inexperienced users would be able to build large agent organisations for real world applications.

An outline of the KARMA/TEAMCORE framework is depicted in Figure A.7. There are two key aspects in the KARMA/TEAMCORE approach. The first focuses on the creation, specification, and monitoring of the agent organisation. The second focuses on enabling the organisation to reliably execute tasks, by ensuring robust teamwork among the agents in the organisation.

KARMA (Knowledgeable Agent Resources Manager Assistant) addresses the first aspect by assisting ABS designers in three ways: First, it provides support for team-oriented programming, where the system designer specifies a hierarchical agent organisation as well as its high level goals, for example to support supply-chain management. Team-oriented programming abstracts away from coordination details, thus eliminating the burden of writing large numbers of coordination plans. Second, KARMA locates agents that match the requirements of the specified organisation and assists in allocating organisational roles to agents. In this way, it alleviates the designer from the burden of searching through vast numbers of agents in the cyberspace. Third, KARMA monitors the agent organisation to diagnose failures and to evaluate agent performance for future reorganisations.

The second aspect of the TEAMCORE approach focuses on robust execution. According to the TEAMCORE approach, the teamwork of agents enhances robust execution, since TEAMCORE agent components are expected to act responsibly towards one another, covering for each other's execution failures and sharing key information. To be able to seamlessly interact with each other, each agent is associated with a TEAMCORE wrapper that is responsible for the interoperability among even heterogeneous agents.

The ABS designer specifies an agent organisation by specifying a *team program*. The team program includes specifications of the organisation hierarchy, the plan hierarchy and the capabilities of agents that could execute those plans. The team organisation hierarchy consists of roles for individuals and for groups of agents. The functionality of the ABS is explicitly expressed by team plans. The developer first assigns roles to plans and then assigns roles to agents. Specifications are done in the STEAM specification language [191].

To locate and recruit agent components, an *agent resources manager* (an analogue of a human resources manager) searches for agents of interest to this organisation in the cyberspace and monitors their performance over time. When a new agent organisation needs to be constructed, KARMA searches different sources, for example yellow pages or other catalogues and compiles a list of different agents together with their properties. From this list, the designer then manually selects the desired agents to include in the organisation. KARMA avoids overwhelming the agent system designer with unnecessary information by including in the list only agents that are capable of playing the roles of the specified organisation.

Concepts	Concept definition	Design in scope	Heuristics support
	\succ	\sqrt	\sqrt
Models	Organisational settings	Collective behaviour	Non-functional aspects
	\sqrt	\sqrt	–
Process	Design perspective	Support for reuse	Design automation
	\downarrow	–	\sqrt
Pragmatics	Generality	Complexity handling	Tool Support
	\oplus	\sqrt	\sqrt

Table A.7: Evaluation of KARMA/TEAMCORE

A.7.2 Evaluation of KARMA/TEAMCORE

KARMA/TEAMCORE is targeting a wide range of agents and application domains. The only restriction therefore is that the agents should exist in the cyberspace in order to participate in the organisation. An ABS in KARMA/TEAMCORE is assembled in a top-down fashion. Furthermore, design heuristics can be specified as rules in the STEAM specification language and taken into account when designing the agent components.

Organisational settings are explicitly modelled in KARMA/TEAMCORE using appropriate roles. Therefore, they can be used as first class design constructs. Collective behaviours are modelled with appropriate team plans, which are assigned to roles. Therefore, collective behaviours can also be used as first class design constructs. Furthermore, some non-functional aspects, for example the performance of the ABS, can be explicitly modelled as constraints in the STEAM specification language. However, there is no comprehensive support for reusing design knowledge.

The KARMA/TEAMCORE approach can be automated to some extent, since a software tool based on the STEAM specifications does the allocation of roles to agents. This tool also assists the designer in specifying team plans and role hierarchies. Furthermore, the formality inherent in the STEAM specification language makes possible for the designer to work at different levels of abstraction with appropriate rigour, reducing therefore the design complexity.

A summary of the evaluation of the KARMA/TEAMCORE approach is given in Table A.7.

A.7.3 Strengths and Weaknesses of KARMA/TEAMCORE

KARMA/TEAMCORE is the most comprehensive ABS engineering approach as far as it concerns supporting the design of the ABSs. It provides assistance to the designers in the

majority of aspects of the evaluation framework proposed in Section 3.1. Notable exceptions are the lack of support for reuse of design knowledge and bottom-up design.

In addition, a major weakness of KARMA/TEAMCORE is that it assumes existing agents, which cannot be generally the case. When suitable agents do not exist, there is no support for creating new agents from scratch. Finally, not all non-functional aspects, for example security, can be conveniently modelled in a quantitative manner as constraints in the STEAM specification language.

Appendix B The Zeus Toolkit

This Appendix describes the Zeus Toolkit components and the Zeus agent realisation process. It also provides details on the Zeus utility agents and the generic Zeus agent structure.

B.1 The Components of the Zeus Toolkit

The ZEUS toolkit consists of three main components: an agent component library, an agent building tool and a suite of utility agents.

B.1.1 The Agent Component Library

This is a package of Java classes that implement the functionality of collaborative agents, that is these classes are the ‘building blocks’ of the agents created during the generation process. This Java library includes classes implementing a number of agent coordination protocols based on the contract-net protocol [184], a number of predefined organizational relationships that can be imposed to agents — such as peer and superior — and a performative-based agent communication language with a comprehensive instruction set. This language was initially based on KQML [65] but in the latest version of the tool the FIPA ACL [66] was implemented as well.

In order to maximise future compatibility, the components of the ZEUS toolkit utilise standardised and low level technology whenever possible. For example, communication takes place through TCP/IP sockets. In the latest versions of the tool, the IIOP and HTTP transport protocols were implemented as well, enabling the creation of Zeus agents capable of connecting to the AgentCities network [203]. AgentCities is an EU Framework V applied research project which aims to demonstrate deployment and interaction of ABSs in a large number of internationally distributed nodes.

The component library also provides full implementations for three types of standard utility agents existing in every ABS developed: the *Agent Name Server*, the *Facilitator* and the *Visualiser*. The utility agents fulfil a support role in the agent society and can be used in any application without modification. The Agent Name Server provides a *white pages* service, matching agent names to network address just like the Domain Name Servers match domain names to IP addresses. The Facilitator provides a *yellow pages* service similar to the UDDI registry; it is used by agents looking for others who are capable of a particular task or service. The role of the Visualiser agent is to provide a pictorial representation of the ABS throughout its execution and it is discussed in the next section.

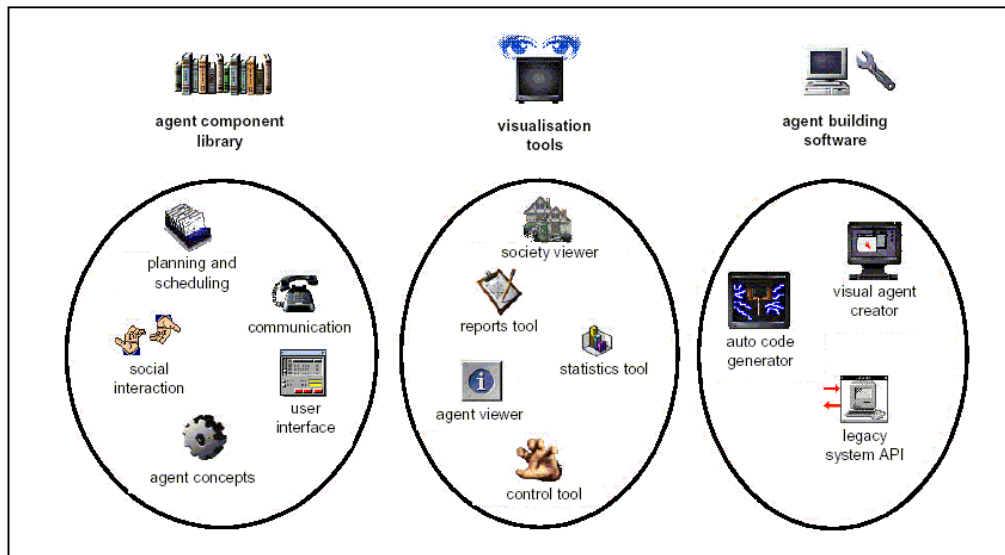


Figure B.1: The components of the Zeus agent building toolkit (Collins et al. 1999)

B.1.2 The Visualisation Tools

The Visualiser tool includes a number of components which attempt to visualise the behaviour of the ABS while it executes. The approach followed is to have a special purpose agent, the Visualiser agent, which is informed of the organisational relations existing between agents and the communication messages exchanged in the ABS.

Monitoring the run-time behaviour of an ABS is not trivial because data, control and active processes are all distributed across the agents. Therefore, the analysis and debugging of ABSs is challenging, as each agent has only a local view of the whole system.

The Visualiser Agent provides a solution to this problem by asking every agent to forward a copy of every message they send to other agents. The messages received can then be collated, interpreted and used to create an up-to-date picture of the agents' collective behaviour. The user interacts with the Visualiser agent through the five Visualisation Tools listed in Figure B.1, with each tool visualising a different aspect of agent society. For instance, the Society Viewer shows all agents known, and the type and frequency of the messages they send, the Reports Tool shows the state of agent tasks and sub-tasks and the Control Tool allows a variety of system housekeeping operations including creating new agents and terminating agent execution. The Society Viewer and the Control Tool are depicted in Figure B.2.

B.1.3 The Agent Building Tools

The agent building tools include the Visual Agent Creator and the Code Generator components and a legacy systems API interfacing Zeus agents with existing Java software. Those tools provide the capability for developing ABSs in Java without the need to know the internal details of how each agent is implemented.

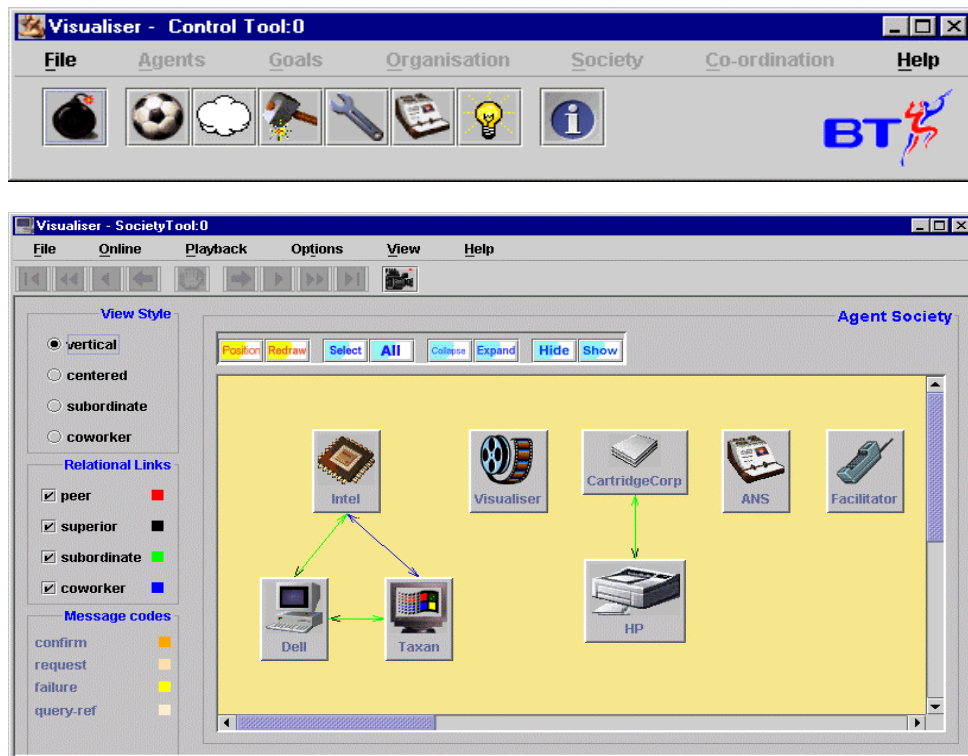


Figure B.2: The Control and Society Tools of the Zeus agent building toolkit

These Visual Agent creator components implement the editors that enable users to interactively create agents by visually specifying their attributes and strategies. A snapshot of the Agent definition interface, depicting the Zeus Agent Generator and the Agent Editor sub-components of the Visual Agent Creator component, is provided in Figure B.3.

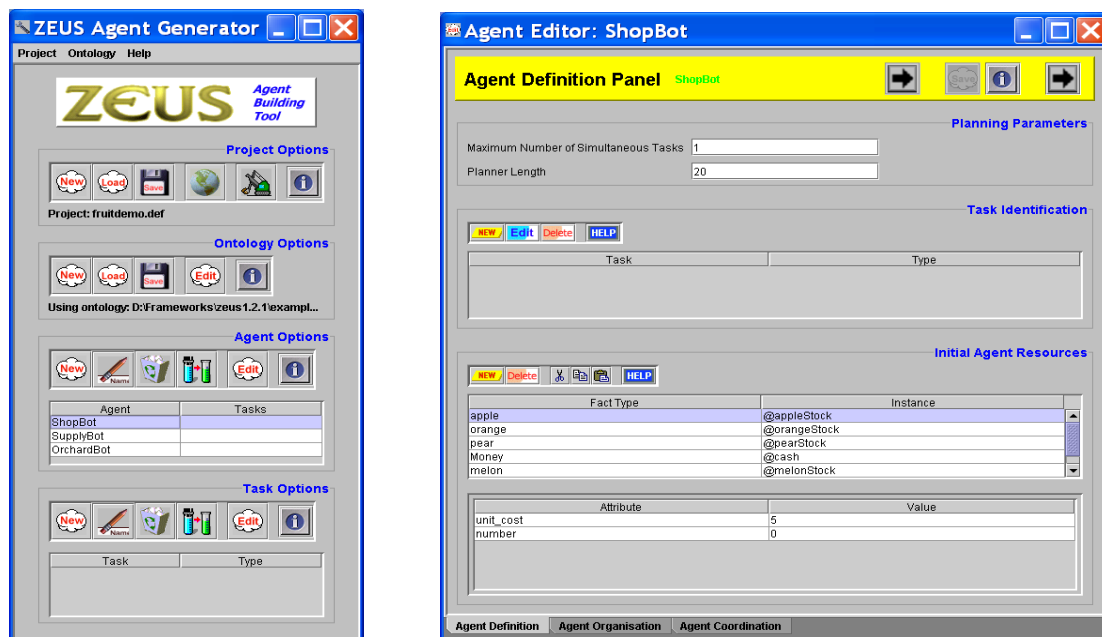


Figure B.3: The agent definition interface of the Zeus agent building tool

In order to generate code for a specific application system, the Code Generator component inherits code from the Agent Component library, and integrates the data from the various visual editors. All Zeus agents are constructed by specialising a generic ZEUS agent (see also Section B.4). The resulting Java program code is then compiled and executed normally. It must be emphasised that the designer does not need to be aware of the detailed internal implementation of each Zeus agent. However, if one has mastered this knowledge, it is possible to modify the Java source code produced by the Code Generator and customise the agents to work in other environments as well.

Existing systems can be linked to the Zeus agents using the Application Programmers' Interface (API) of the wrapper class that is also part of the toolkit. The developer describes the intended agents with the Agent Creation tools and the Code Generator generates Java source code using classes from the Agent component library. Once their tasks have been implemented the agents can be executed, and observed using the visualisation tools provided. Using the above tools together substantially facilitates the engineering of intelligent collaborative ABSs.

In order for agents to be able to understand the common domain concepts they need to be aware of the same ontology of shared concepts. This is supported in Zeus by the Ontology Editor, which is part of the Visual Agent creation tool. The Ontology editor supports definition, storing and retrieving of ontologies which can then be used by different ABSs. A snap shot of the ontology editor is provided in Figure B.4.

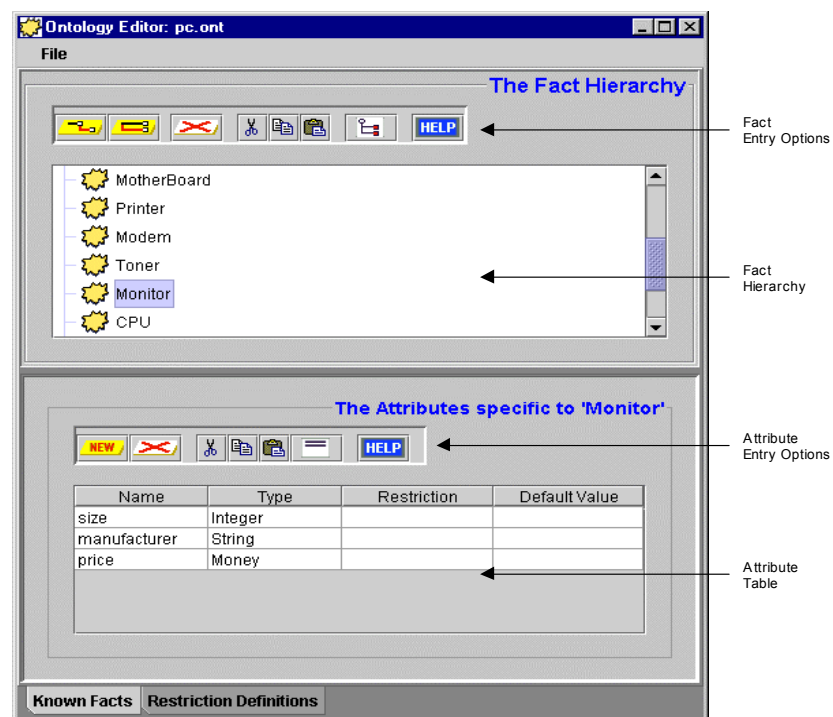


Figure B.4: The Ontology editor

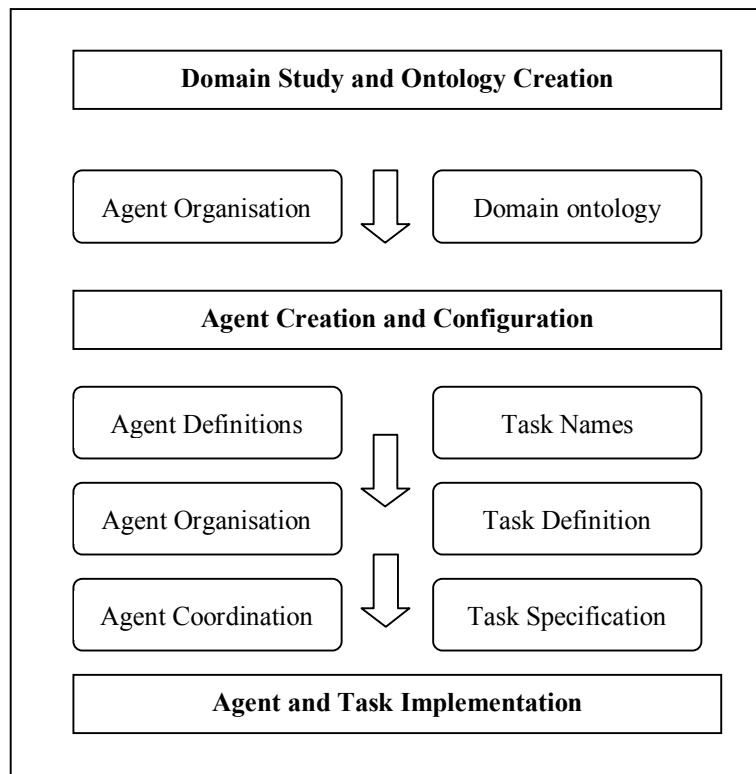


Figure B.5: The Zeus agent realisation process (Nwana et al. 1999)

B.2 The Zeus Agent System Realisation Process

The Zeus agent development methodology includes analysis, design, realisation and run-time support stages. The agent realisation process refers to the design and code generation for the agents of the ABS. In particular, it refers to creating an ontology, defining the goals, tasks and initial resources (facts) of the agents and setting the run-time parameter for utility and agents.

At the highest level of abstraction, the ZEUS agent development approach requires developers to view an agent as composed of three layers: a *definition layer*, an *organisation layer* and a *coordination layer*. At the definition layer the agent is viewed as an autonomous reasoning entity in terms of its competencies, rationality model, resources, beliefs, and preferences. At the organisation layer it is viewed in terms of its relationships with other agents, for example what other agents it is aware of, and what abilities it knows they possess. At the co-ordination layer the agent is viewed as a social entity in terms of its co-ordination and negotiation techniques. This agent model is supplemented with the protocols that implement inter-agent communication and an application programmer's interface that enables the agent to be linked to the external programs that provide it with resources and/or implement its competencies.

The realisation process consists of the following stages (Figure B.5):

Stage 1: Ontology Creation: The first stage in the agent realisation process is to define an appropriate application ontology representing the significant concepts, attributes and values

within the application domain. The contents of the ontology consist of *fact types* each one associated with a number of *attributes* and *fact instances*, or *facts*, which have values assigned to their attributes. Fact types can be derived from other fact types via inheritance in a manner similar to object-oriented programming. The ontology definition is done using the ZEUS Ontology Editor mentioned in Section B.1.3. Alternatively, an existing ontology can be imported.

Stage 2: Agent Creation and Configuration: During this stage the generic ZEUS agent is configured to fulfil its application-specific responsibilities, resulting in a number of application agents (also called *task agents* in Zeus terminology). This stage is carried out using the ZEUS Agent Editor and it involves four sub-stages:

1. *Agent Definition* - where the tasks, initial resources and planning abilities of the agent are specified. This involves specifying the number of tasks that an agent can carry out concurrently and the time frame in which the agent will plan its activities. The default values are 1 and 20 respectively. Furthermore, at this stage the resources⁹ initially available to the agent are specified. Finally, agent definition involves specifying the types of tasks the agent is capable of carrying out. Zeus currently supports three types of tasks, namely *primitive*, *rulebase*, *planscripts* and *summary* tasks. Detailed specification of the functionality of each task is done in the next sub-stage.
2. *Task Description* - where the functionality and attributes of agent tasks are specified. Primitive tasks are developed as external Java classes invoked by the agent when required. Rulebase tasks are developed as CLIPS-like rules executed each time by the built-in agent rule engine. Summary tasks are defined as combinations of primitive and rulebase tasks. However, summary tasks are not supported in the current version of Zeus. Task execution is triggered when appropriate events are perceived by the agent. This is done when a number of *task preconditions* are met and it results to a number of *task effects*. The flow of information between agents and tasks on task execution is depicted in Figure B.6.
3. *Agent Organisation* - where the organisational relationships of each agent are specified. Zeus currently supports four types of organisational relationships:
 - *Peer* - which is the default relationship an agent has with other agents.

⁹ In Zeus the resources of an agent correspond to facts stored to the agent's knowledge base. Facts are parts of the common ontology used by the agents and can be entered or removed dynamically throughout the lifecycle of the agent.

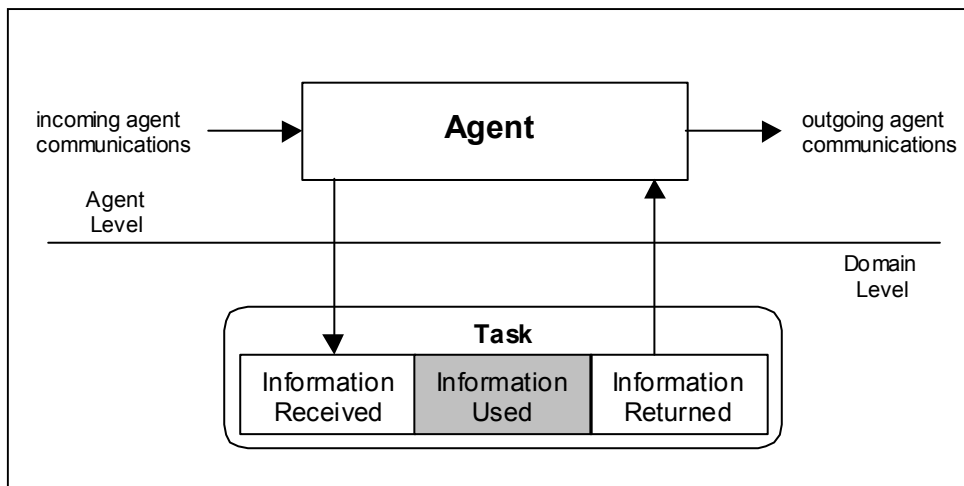


Figure B.6: The flow of information between an agent and a task

- *Superior* - which means that an agent has authority to require tasks to be carried out by certain other agents.
- *Subordinate* - meaning that the agent is obliged to carry out tasks when required by certain other agents (which are its superiors).
- *Co-worker* – indicating that there is a social relationship between certain agents. Co-workers are contacted after subordinates and before peers when there is a need for an agent to outsource one of its tasks.

Furthermore, at this stage the *abilities* of each acquaintance, namely the tasks that it is possible to carry out, are specified.

4. *Agent Co-ordination* - where each agent is equipped with coordination protocols and negotiation strategies. Currently only the contract net coordination protocol and two simple negotiation strategies, linear and exponential decay are supported. However, it is possible for the ABS designer to implement proprietary coordination protocols and negotiation strategies.

Stage 3: Agent and Task Implementation: Here the run-time parameters of the utility and application agents are specified and the agent Java source code is generated. The agent tasks are implemented manually by the agent engineers. In particular, this stage involves the following sub-stages:

1. *Utility Agent Configuration:* This involves defining the attributes of the utility agents who provide the support infrastructure for the agent society. This information is entered through the Code Generation Editor and it is used for the creation of the utility agents.

2. *Application Agent Configuration*: Agent configuration involves specifying the runtime parameters of the application agents. This requires supplying information such as the host machines the agents will run on, and the external resources and programs to which the agents will be linked.
3. *Agent code generation and task implementation*: Finally, the agent source code is automatically generated via the Code Generator component. This leaves the developer with the job of providing the application-specific implementations of the tasks, external resources, programs (such as agent user interfaces) and interaction strategies. When this stage has been completed the ABS is ready for deployment.

The agent realisation process is the part of the Zeus agent development methodology which has been extended to support RAMASD (see Section 6.3). The Zeus agent development methodology is critically discussed in Appendix A.6.

B.3 The Zeus Utility Agents

The Zeus agent building toolkit follows the FIPA standards regarding agent system management [67]. Each Zeus ABS is assumed to operate in a separate agent platform¹⁰. In particular, each Zeus ABS includes three utility agents, the *Agent Name Server (ANS)*, the *Facilitator* the *Agent Communication Channel (ACC)*, and an arbitrary number of application agents (see Figure B.7). The utility agents serve the purpose of enabling the application agents to find about services that other agents in the agent system offer and to facilitate inter-agent and inter-platform communication. More specifically the responsibilities of the utility agents are the following:

- *Agent Name Server*: The Agent Name Server provides white pages lookup services to the other agents. Every agent, when it is first executed it contacts the ANS and its internal address is stored together with its name in an ANS database. Any agent wishing to contact another agent it hasn't contacted previously, requests details about the new agent's address from ANS. Subsequently, communication is initiated via an appropriate TCP/IP socket.
- *Facilitator*: Each Zeus agent aims to fulfil certain goals and is able to carry out certain *tasks* (see also Section A.6). Those tasks are termed *capabilities* in the Zeus terminology and can be carried out to fulfil the agent's goals or on behalf of other agents upon request. In the

¹⁰ According to the FIPA standards an *Agent Platform (AP)* consists of a number of host computers, operating system, agent support software, an inter-agent communication method termed *Message Transport System (MTS)*, two FIPA agent management logical components (usually implemented as utility agents): *Directory Facilitator (DF)*, and *Agent Management System (AMS)* and an arbitrary number of application agents.

latter case, they are considered as *services* that the agent provides to other agents. The Facilitator agent serves as a yellow pages directory listing all services that agents in the ABS are able to provide to others.

On system initialisation, all agents register their capabilities with the Facilitator. When an agent is unable to complete all tasks required to fulfil its goals alone, it considers seeking assistance from other agents in the ABS. In that case, it contacts the Facilitator agent and if there is another agent capable of carrying out this particular task the Facilitator agent provides its address for direct interaction.

- *Agent Communication Channel (ACC)*: The ACC agent is the Agent Communication Channel as specified by FIPA [67]. It can be thought of as the single point of contact for all agents on a platform. The basic function of the ACC is to forward messages to the other agents, and to accept messages for forwarding from agents on it's platform and pass them on. The ACC agent in the current release of Zeus (1.2.1) supports FIPA'97 and FIPA 2000 IIOP transport and FIPA 2000 HTTP support implemented by sockets that listen over TCP/IP for connections from remote machines.

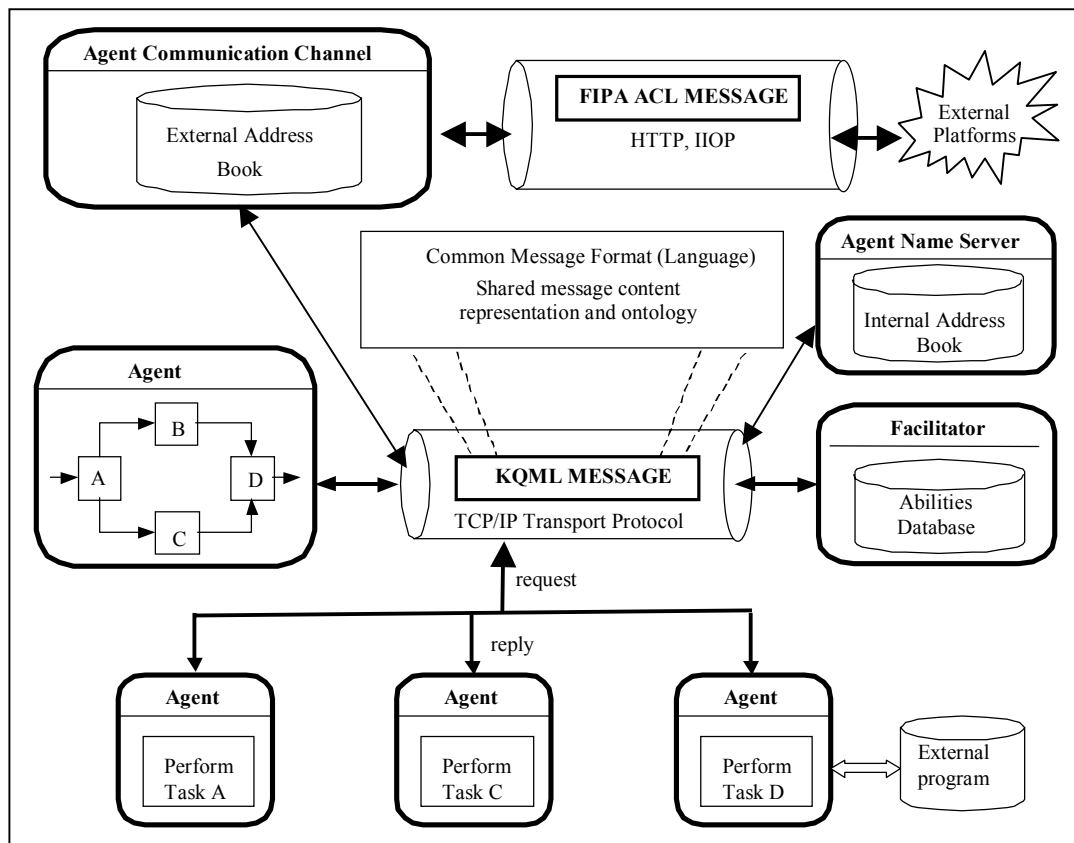


Figure B.7: The Zeus ABS structure (Thompson 2001)

The ACC agent also provides translation facilities between Zeus communication language messages and FIPA ACL messages, in particular resolving FIPA addresses to Zeus addresses (and vice versa) and providing FIPA envelopes. For example, when a message from outside Zeus is received by the ACC agent, it generates a new alias for the external agent, registers the alias in the name server, translates the incoming message to Zeus performatives and sends the message to the appropriate agent. A similar procedure is followed when a Zeus agents wants to communicate with an external agent.

Zeus is an evolving software package and therefore certain agent platform characteristics which have recently become FIPA standards have not been implemented yet. For example, Zeus still communicates with the external world with a special purpose utility agent, the Agent Communication Channel. This term was used in FIPA 98 specifications while in FIPA 2000 it has been replaced by the term Agent Management Service. Furthermore, FIPA AMS and FIPA DF are not currently supported [198]. However, there is an active Zeus users community [197], which constantly updates the Zeus tool. Therefore, it is expected that Zeus will be fully compatible with all current FIPA specifications in the near future.

B.4 The Generic Zeus Agent

The components of the Agent Component Library enable the construction of an application independent generic ZEUS agent that can be customised for specific applications by imbuing it with problem-specific resources, competencies, information, organisational relationships and co-ordination protocols. The generic ZEUS agent internal structure includes the following components (Figure B.8):

- a *Mailbox* that handles communications between the agent and other agents.
- a *Message Handler* that processes incoming messages from the Mailbox, dispatching them to the relevant components of the agent.
- a *Co-ordination Engine* that makes decisions concerning the goals of the agent, for example how they should be pursued and when to abandon them. The coordination engine is also responsible for co-ordinating the agent's interactions with other agents using known co-ordination protocols and strategies. In the current version of Zeus only the contract net protocol has been implemented.
- an *Acquaintance Database* that contains information regarding the relationships of the agent and other agents in the ABS. In addition, the Acquaintance database contains the beliefs the agent has about the capabilities of its acquaintance agents. The Co-ordination Engine uses information contained in this database when making collaborative arrangements with other agents.

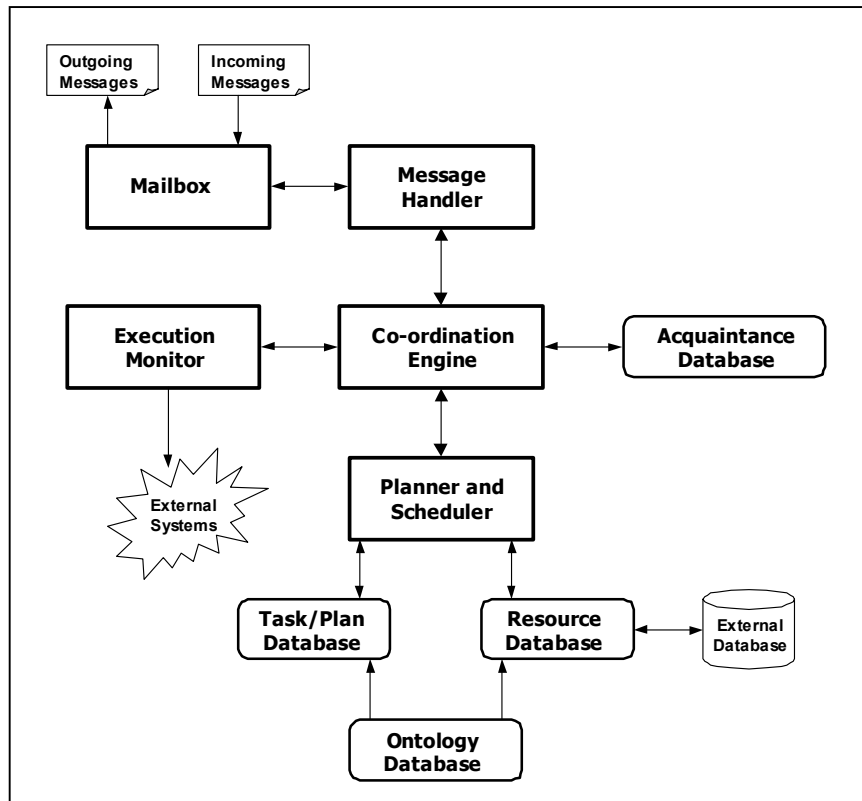


Figure B.8: The generic Zeus agent internal structure (Collins and Ndumu 1999)

- a *Planner and Scheduler* component that plans the tasks of the agent based on decisions taken by the Co-ordination Engine and the resources and task specifications available to the agent.
- a *Resource Database* that maintains a list of resources (referred to in this paper as *facts*) that are owned by and available to the agent. The Resource Database also supports a direct interface to external systems, in other words it is possible to programmatically manipulate the resource database of an agent from external Java classes.

Appendix C RCL EBNF Syntax

The EBNF syntax of RCL is the following:

```
/* EBNF_production section */

RclProgram : "Program" <IDENTIFIER> "{" ( RclProduction )* "}" <EOF>

RclProduction : RoleDeclaration | Statement | RoleConstraints

RoleConstraints : <IDENTIFIER> <ROP> <IDENTIFIER> ";"

RoleDeclaration : DeclarationSpecifiers [ InitDeclaratorList ] ";"

DeclarationSpecifiers : TypeSpecifier [ DeclarationSpecifiers ]

TypeSpecifier : <STRING> | <INT> | <REAL> | RoleSpecifier

RoleSpecifier : <ROLE> ([ <IDENTIFIER> ] "{" RoleParamDeclaration "}" |
<IDENTIFIER> )

RoleParamDeclaration : (SpecifierQualifierList RoleDeclaratorList ";")+

SpecifierQualifierList : TypeSpecifier [ SpecifierQualifierList ]

RoleDeclaratorList : RoleDeclarator ( "," RoleDeclarator )*

RoleDeclarator : Declarator | [ Declarator ] ":" ConstantExpression

InitDeclaratorList : InitDeclarator ( "," InitDeclarator)*

InitDeclarator : Declarator [ <EQUALS> Initializer ]

Declarator : (<IDENTIFIER> | "(" Declarator ")") ( "[" [ ConstantExpression ]
                "]" | "(" ParameterTypeList ")" | "(" [ IdentifierList ] ")" )*

ParameterTypeList : ParameterList [ "," "..." ]

ParameterList : ParameterDeclaration ( "," ParameterDeclaration)*

ParameterDeclaration : DeclarationSpecifiers Declarator

IdentifierList : <IDENTIFIER> ( "," <IDENTIFIER>)*

Initializer : AssignmentExpression | "{" InitializerList() [ "," ] "}"

InitializerList : Initializer ( "," Initializer())*

TypeName : SpecifierQualifierList


/* Statements */

Statement : ExpressionStatement | CompoundStatement

ExpressionStatement : [ Expression ] ";"

CompoundStatement : "{" [ DeclarationList ] [ StatementList ] "}"

DeclarationList : RoleDeclaration+
```

```

StatementList :      (Statement)+

/* Expressions */

Expression : AssignmentExpression ( "," AssignmentExpression() ) *
AssignmentExpression: PostfixExpression <EQUALS> AssignmentExpression
                    | ConditionalExpression
ConstantExpression : ConditionalExpression
ConditionalExpression : LogicalORExpression
LogicalORExpression: LogicalANDExpression [ "|" LogicalORExpression]
LogicalANDExpression: EqualityExpression [ "&&" LogicalANDExpression]
EqualityExpression : RelationalExpression [ ( "==" | "!=" )
                    EqualityExpression ]
RelationalExpression : AdditiveExpression [ ( "<" | ">" | "<=" |
                    ">=" ) RelationalExpression ]
AdditiveExpression : MultiplicativeExpression [ ( "+" | "-" )
                    AdditiveExpression ]
MultiplicativeExpression : PostfixExpression [ ( "*" | "/" | "%" )
                    MultiplicativeExpression ]
PostfixExpression : PrimaryExpression ( "(" [ArgumentExpressionList
                    ] ")" | "." <IDENTIFIER> ) *
PrimaryExpression : <IDENTIFIER> Constant
ArgumentExpressionList: AssignmentExpression ( ","
                    AssignmentExpression ) *
Constant: <INTEGER_LITERAL> | <FLOATING_POINT_LITERAL> | <STRING_LITERAL>

/* ROLE CONSTRAINT SYMBOLS */

ROP: <NOT> | <AND> | <ADD> | <EQ> | <MERGE> | <IN>

NOT: "not" | "NOT"
AND: "and" | "AND"
ADD: "add" | "ADD"
EQ: "eq" | "EQ"
MERGE: "merge" | "MERGE"
IN: "in" | "IN"

```

```

/* KEYWORDS */

INT: "int"

STRING: "string"

REAL: "real"

ROLE: "Role"


/* SYMBOLS */

EQUALS:      "="

LBRACE:      "{"

RBRACE:      "}"


/* IDENTIFIERS */

IDENTIFIER: <LETTER> (<LETTER>|<DIGIT>)*

LETTER:      [ "a"-"z", "A"-"Z" ]

DIGIT: [ "0"-"9" ]

INTEGER_LITERAL: [ "1"-"9" ] ([ "0"-"9" ])*

FLOATING_POINT_LITERAL: ([ "0"-"9" ])+ "." ([ "0"-"9" ])*

STRING_LITERAL:      "\"" (~["\"", "\\", "\n", "\r"] | "\"\"")
([ "n", "t", "b", "r", "f", "\\", "\'", "\""] | [ "0"-"7" ] ([ "0"-"7" ])? | [ "0"-"3" ]
[ "0"-"7" ] [ "0"-"7" ]))* "\""

```


References

1. Alagar, V.S. and K. Periyasamy, *Specification of Software Systems*. Graduate Texts in Computer Science, ed. D. Gries and F.B. Schneider. 1998, New York: Springer-Verlag, p. 422, ISBN: 0-387-98430-5.
2. Andersen, E.P., *Conceptual Modelling of Objects: A Role Modelling Approach*. PhD Thesis, *Dept of Computer Science*. 1997, University of Oslo: Oslo, Norway. p. 333.
3. Andrews, P. and S. Adler, *Privacy: Basing Service on Respect*. 2001, Center for IBM E-business Innovation, IBM Labs: New York, NY. p. 4.
4. Antonsson, E.K. and J. Cagan, eds. *Formal Engineering Design Synthesis*. 2001, Cambridge University Press: Cambridge, p. 450, ISBN: 0521792479.
5. Bansiya, J. and C.G. Davis, *A hierarchical model for object-oriented design quality assessment*. IEEE Transactions on Software Engineering, 2002. **28**(1): p. 4-17.
6. Banton, M., *Roles: An introduction to the study of social relations*. 1965, London: Tavistock Publications, p. 224, ISBN: 0422721204.
7. Barber, K.S., T.H. Liu, and D.C. Han, *Agent-Oriented Design*. 1999, University of Texas at Austin: Austin, TX, USA. p. 14.
8. Barber, K.S., A. Goel, and C.E. Martin, *The Motivation for Dynamic Adaptive Autonomy in Agent-based Systems*. 1999, University of Texas at Austin: Austin, TX, USA. p. 11.
9. Barber, K.S., D.C. Han, and T.-H. Liu, *Strategy Selection-Based Meta level Reasoning for Multi-Agent Problem Solving*, in *Agent-Oriented Software Engineering*, P. Ciancarini and M. Wooldridge, eds, 2001, Springer-Verlag: Berlin, p. 269-283, ISBN 3-540-41594-7.
10. Bartelt, A. and W. Lamersdorf, *Agent-Oriented Concepts to Foster the Automation of e-Business*, in *Proceedings of the 11th International Workshop on Database and Expert Systems (DEXA 2000)*, A.M. Tjoa, R.R. Wagner, and A. Al-Zobaidi, eds. 2000, IEEE Computer Society Press: Munich, Germany, p. 775-779, ISBN 0-7695-0680-1.
11. Basin, D. and S. Friedrich, *Modeling a Hardware Synthesis Methodology in Isabelle*. Formal Methods in System Design, 1999. **15**(2): p. 99-122.
12. Bauer, B., J.P. Muller, and J. Odell, *Agent UML: A Formalism for Specifying Multiagent Software Systems*, in *Agent-Oriented Software Engineering*, P. Ciancarini and M. Wooldridge, eds, 2001, Springer-Verlag: Berlin, p. 106-119, ISBN 3-540-41594-7.
13. Bellifemine, F., A. Poggi, and G. Rimassa. *JADE - a FIPA-compliant agent framework*. in *PAAM 99*. 1999. London, UK.
14. Bertino, E., E. Ferrari, and V. Atluri. *RBAC support in object-oriented role databases*. in *ACM Workshop on Role-based Access Control*. 1997. Fairfax, VA USA: ACM Press.
15. Biddle, B.J., *Role Theory: Expectations, Identities and Behaviors*. 1979, London: Academic Press, p. 416, ISBN 0-12-095950-X.
16. Biddle, B.J. and E. J. Thomas, eds, *Role Theory: Concepts and Research*. 1979, Huntington, New York: Robert E. Krieger Publishing Company, ISBN 0-47-107215-X.
17. Blake, M.B., *A Development Approach for Workflow-Based E-Commerce using Reusable Distributed Components*, in *2000 Americas Conference on Information*

Systems (AMCIS2000) (Track on Workflow Technology and E-Commerce Applications). 2000: Long Beach, CA.

18. Blake, M.B., *Innovations in Software Agent-Based B2B Technologies*, in *Workshop on Agent-Based Approaches to B2B at the Fifth International Conference on Autonomous Agents (AGENTS 2001)*. 2001, ACM Press: Montreal, Canada. p. 1-7.
19. Blake, M.B., *Towards the Use of Agent Technology for B2B Electronic Commerce*. 2002: Department of Computer Science, Georgetown University, 234 Reiss Science Building, Washington. p. 16.
20. Brazier, F., et al., *Formal Specification of Multi-Agent Systems: a Real-World Case*, in *First International Conference on Multi-Agent Systems (ICMAS'95)*. 1995: San Francisco, CA. p. 25-32.
21. Brazier, F.M.T., P.A.T.V. Eck, and J. Treur, *Modelling a Society of Simple Agents: From Conceptual Specification to Experimentation*. Applied Intelligence, 2001. **14**(2): p. 161-178.
22. Brazier, F.M.T., et al., *DESIRE: Modelling Multi-Agent Systems in a Compositional Formal Framework*. International Journal of Cooperative Information Systems, Special Issue on Formal Methods in Cooperative Information Systems: Multi-Agent Systems, 1997. **5**(1): p. 67-94.
23. Bresciani, P., et al. *A Knowledge Level Software Engineering Methodology for Agent Oriented Programming*. in *Proceedings of the Fifth International Conference on Autonomous Agents (AGENTS 2001)*. 2001. Montreal, Canada: ACM Press.
24. Bresciani, P., et al., *Modeling Early Requirements in Tropos: a Transformation Based Approach*, in *Agent-Oriented Software Engineering II, Second International Workshop (AOSE 2001), Montreal, Canada*, M.J. Wooldridge, G. Weis, and P. Ciancarini, eds. 2002, Springer Verlag: Berlin, p. 151-168, ISBN 3-540-43282-5.
25. Briand, L.C. and J. Wust, *Modeling development effort in object-oriented systems using design properties*. IEEE Transactions on Software Engineering, 2001. **27**(11): p. 963 - 986.
26. Briand, L.C., S. Morasca, and V.R. Basili, *Property-based software engineering measurement*. IEEE Transactions on Software Engineering, 1996. **22**(1): p. 68 - 86.
27. Bubler, C. *Capability-based modelling*. in *First International Conference on Enterprise Integration Modelling*. 1992. Carolina: MIT Press.
28. Busbach, U., *Activity Coordination in Decentralised Working Environments*, in *Remote Cooperation: CSCW Issues for Mobile and Teleworkers*, A. Dix and R. Beale, eds. 1996, Springer-Verlag: Berlin, p. 95-112, ISBN 3-540-76035-0.
29. Caire, G., F. Leal, and J. Rodriguez, *MESSAGE: Methodology for Engineering Systems of Software Agents*, in *Recommendations on supporting tools*. 2001, EURESCOM: Heidelberg. p. 27.
30. Caire, G., et al., *Agent Oriented Analysis Using Message/UML*, in *Agent-Oriented Software Engineering II, Second International Workshop, (AOSE 2001), Montreal, Canada*, M.J. Wooldridge, G. Weis, and P. Ciancarini, eds. 2002, Springer Verlag: Berlin, p. 151-168, ISBN 3-540-43282-5.
31. Castro, J., M. Kolp, and J. Mylopoulos. *Developing Agent-Oriented Information Systems for the Enterprise*. in *Second International Conference On Enterprise Information Systems*. 2000. Stafford, UK.
32. Chandrasekaran, B., T. Johnson, and J.W. Smith, *Task Structure Analysis for Knowledge Modeling*. Communications of the ACM, 1992. **33**(9): p. 124-136.

33. Chatha, K.S. and R. Vemuri, *An Iterative Algorithm for Hardware-Software Partitioning, Hardware Design Space Exploration and Scheduling*. Design Automation for Embedded Systems, 2000. **5**(3-4): p. 281-293.
34. Chuang, T.-T. and S.B. Yadav. *An Agent-Based Architecture of an Adaptive Decision Support System*. in *Proceedings of Second Americas Conference on Information Systems*. 1997.
35. Chung, L., et al., *Non-Functional Requirements in Software Engineering*. The Kluwer International Series in Software Engineering, V.R. Basili. ed. 2000, New York, NY: Kluwer Academic Publishers, ISBN 0-7923-8666-3.
36. Cockburn, A., *Structuring Use Cases with Goals*. 1997, SIGS Publications: Journal of Object-Oriented Programming.
37. Cohen, P.R. and H.J. Levesque, *Intention is choice with commitment*. Artificial Intelligence, 1990. **42**: p. 213-261.
38. Collins, J. and D. Ndumu, *The Zeus Agent Building Toolkit: The Role Modelling Guide*. 1999.
39. Coplien, J.O., *A Generative Development Process Pattern Language*, in *Pattern Languages of Program Design*, J.O. Coplien and D.C. Schmidt, eds. 1995, Addison-Wesley: New York, p. 183-237, ISBN 0-201-60734-4.
40. Coplien, J.O., *The Column Without a Name: After all, we can't ignore efficiency*, in *C++ Report*. 1996. p. 71-74.
41. Coplien, J.O. and D.C. Schmidt, eds. *Pattern Languages of Program Design*. 1995, Addison-Wesley: New York, p. 562, ISBN 0-201-60734-4.
42. Covisint, *Covisint Solution Suites*. 2002, Covisint, LLC.
43. Coyne, R.D., et al., eds. *Knowledge-based design systems*. 1990, New York: Addison-Wesley, ASIN: 0201103818.
44. Dascalu, S. and P. Hitchcock, *An Approach to Integrating Semi-formal and Formal notations in Software specifications*, in *Proceedings of the 2002 ACM Symposium on Applied Computing*. 2002, ACM Press: Madrid, p. 1014-1020.
45. Davis, J.S. and R.J. LeBlanc, *A study of the applicability of complexity measures*. IEEE Transactions on Software Engineering, 1988. **14**(9): p. 1366-1372.
46. Decker, K.S., *Environment Centered Analysis and Design of Co-Ordination Mechanisms*. 1995, PhD Thesis, University of Massachusetts at Amherst.
47. Dellarocas, C. and M. Klein, *Civil Agent Societies: Tools for Inventing Open Agent-Mediated Electronic Marketplaces*, in *Agent Mediated Electronic Commerce II, Towards Next-Generation Agent-Based Electronic Commerce Systems, IJCAI 1999 Workshop*, A. Moukas, C. Sierra, and F. Ygge, eds. 2000, Springer Verlag: Berlin, p. 24-39, ISBN 3-540-67773-9.
48. DeLoach, S.A., *Modeling Organizational Rules in the Multi-agent Systems Engineering Methodology*, in *Proceedings of the 15th Canadian Conference on Artificial Intelligence (AI'2002). Calgary, Alberta, Canada. May 27-29, 2002*, R. Cohen and B. Spencer, eds. 2002, Springer Verlag: Berlin, Heidelberg, p. 1-15, ISBN 3-540-43724-X.
49. DeLoach, S.A., M.F. Wood, and C.H. Sparkman, *Multi-Agent Systems Engineering*. International Journal of Software Engineering and Knowledge Engineering, 2001. **11**(3): p. 231-258.
50. Depke, R., R. Heckel, and J.M. Kuster, *Agent-Oriented Modelling with Graph Transformation*, in *Agent-Oriented Software Engineering I, First International*

- Workshop (AOSE 2000), Limerick, Ireland*, P. Ciancarini and M. Wooldridge, eds. 2001, Springer-Verlag: Berlin, p. 106-119, ISBN 3-540-41594-7.
51. Depke, R., R. Heckel, and J.M. Kuster, *Improving the Agent-oriented Modeling Process by Roles*, in *Proceedings of the fifth international conference on Autonomous Agents*. 2001, ACM Press: Montreal, Canada.
 52. Depke, R., R. Heckel, and J.M. Kuster, *Formal Agent-Oriented Modeling with Graph Transformation*. Science of Computer Programming, 2001.
 53. Durante, A., et al., *A Model for the E-Service Marketplace*. 2000, HP Laboratories: Palo Alto, CA. p. 22.
 54. Eden, A.H., *Precise Specification of Design Patterns and Tool Support in Their Application*, in *Department of Computer Science*. 2000, PhD Thesis, Tel Aviv University: Tel Aviv. p. 101.
 55. Eiter, T. and V. Mascardi, *Comparing Environments for Developing Software Agents*. AI Communication, 2002.
 56. Elammari, M. and W. Lalonde. *An Agent-Oriented Methodology: High-Level and Intermediate Models*. in *CaiSE Workshop on Agent Oriented Information Systems (AOIS'99)*. 1999. Heidelberg: MIT Press.
 57. Electronic Marketplaces Source Guides, *Business to Business Marketplaces to the Automotive Industry*. 2000, Momentum Technologies LLC.
 58. Eles, P., et al., *System Level Hardware/Software Partitioning Based on Simulated Annealing and Tabu Search*. Design Automation for Embedded Systems, 1997. 2(1): p. 5-32.
 59. Elliot, J., *A General Theory of Bureaucracy*. 1976, London, UK: Heinemann Press, ISBN 0-435-824-732.
 60. Evans, R., et al., *MESSAGE: Methodology for Engineering Systems of Software Agents*, in *Methodology for Agent-Oriented Software Engineering*. 2001, EURESCOM: Heidelberg. p. 75.
 61. Fenton, N. and S.L. Pfleeger, *Software Metrics: A Rigorous and Practical Approach*. Second ed. 1997, Boston, MA, USA: PWS Publishing Co, p. 656, ISBN 0-534-95425-1.
 62. Ferber, J., *Multi-Agent Systems: An Introduction to Distributed Artificial Intelligence*. 1999, Singapore: Addison Wesley, p. 509, ISBN 0-201-36048-9.
 63. Ferber, J. and O. Gutknecht. *A meta-model for the analysis and design of organisations of Multi-Agent systems*. in *Proceedings of the International Conference in Multi-Agent Systems (ICMAS 98)*. 1998. Paris, France: IEEE Computer Society Press.
 64. Ferber, J., et al. *Organization Models and Behavioral Requirements Specification for Multi-Agent Systems*. in *The Fourth International Conference on MultiAgent Systems (ICMAS-2000)*. 2000. Boston MA, USA.: IEEE Press.
 65. Finin, T., Y. Lambrou, and J. Mayfield, *KQML as an Agent Communication Language*, in *Software Agents, chap. 14*, J.M. Bradshaw, ed. 1997, MIT Press: New York, p. 291-316, ISBN 0-262-52234-9.
 66. FIPA98, *FIPA 98 Specification*. <http://www.fipa.org>, 1998.
 67. FIPA, *FIPA 2000 Specification*. 2000.
 68. Fisher, M. and M. Wooldridge, *On the formal specification and verification of Multi-Agent Systems*. International Journal of Cooperative Information Systems, 1997. 6(1): p. 37-65.

69. Fox, M.S., M. Barbuceanau, and R. Teigen, *Agent-Oriented Supply-Chain Management*. The International Journal of Flexible Manufacturing Systems, 2000. **12**: p. 165-188.
70. Franklin, S. and A. Graesser, *Is It an Agent, or Just a Program?: A Taxonomy for Autonomous Agents*, in *Agent Theories, Architectures and Languages III*. 1996, Springer Verlag: Berlin, p. 193-206, ISBN 3-540-62507-0.
71. Gajski, D.D., et al., *High-level synthesis: Introduction to chip and system design*. 1992, Boston: Kluwer Academic Publishers, p.384, ISBN 0-7923-9194-2.
72. Galbraith, J., *Organisation Design*. 1977, Reading, MA: Addison Wesley, p.426, ISBN 0-2010-2558-2.
73. Gasser, L., *Perspectives on Organizations in Multi-agent Systems*, in *Multi-Agent Systems and Applications, 9th ECCAI Advanced Course ACAI 2001 and Agent Link's 3rd European Agent Systems Summer School, EASSS 2001, Prague, Czech Republic, July 2-13, 2001. Selected Tutorial Papers*, M. Luck, et al., eds. 2001, Springer Verlag, p. 1-16, ISBN 3-540-42312-5.
74. Genesereth, M.R. and N. Nilsson, *Logical Foundations of Artificial Intelligence*. 1987, San Mateo, CA: Morgan Kaufmann, p. 405, ISBN 0-9346-1331-1.
75. Genesereth, M.R. and S.P. Ketchpel, *Software Agents*. Communications of the ACM, 1994. **37**(7): p. 48-53.
76. Gerber, C. *Bottleneck Analysis as a Heuristic for Self-Adaption in Multi-Agent Societies*. in *Joint Conference on the Science and Technology of Intelligent Systems ISIC/CIRA/ISAS*. 1998. GAITHERSBURG, MARYLAND: IEEE Control Systems Society Press.
77. Giunchiglia, F., J. Mylopoulos, and A. Perini, *The Tropos Software Development Methodology: Processes, Models and Diagrams*. 2002, ITC - IRST. in *Autonomous Agents and Multi Agent Systems (AAMAS)*, 2002, Bologna, Italy, ACM Press.
78. Glaser, N., *The CoMoMAS Methodology and Environment for Multi-Agent System Development*, in *Multi-Agent Systems: Methodologies and Applications, Second Australian Workshop on Distributed Artificial Intelligence, Cairns, Queensland, Australia, August 27, 1996, Revised Papers*, C. Zhang and D. Lukose, eds. 1997, Springer Verlag: Berlin, p. 1-16, ISBN 3-540-63412-6.
79. Glaser, N. and P. Morignot, *The Reorganization of Societies of Autonomous Agents*, in *Multi-Agent Rationality: 8th European Workshop on Modelling Autonomous Agents in a Multi-Agent World, MAAMAW'97, Ronneby, Sweden, May 13-16, 1997, Proceedings*, M. Boman and W.V.d. Velde, eds. 1997, Springer Verlag: Berlin, p. 98-111, ISBN 3-540-63077-5.
80. Greenspan, S., J. Mylopoulos, and A. Borgida, *On formal requirements modeling languages: RML revisited*, in *Proceedings of the 16th International Conference on Software Engineering*. 1994, IEEE Computer Science Press. p. 135-148.
81. Griss, M.L. and G. Pour, *Accelerating Development with Agent Components*. IEEE Computer, 2001. **34**(5): p. 37-43.
82. Guttman, R.H., A.G. Moukas, and P. Maes, *Agent-mediated Electronic Commerce: A Survey*. Knowledge Engineering Review, 1998. **13**(2): p. 147-159.
83. Haddadi, A. and K. Sundermeyer, *Belief-Desire-Intention Agent Architectures*, in *Foundations of Distributed Artificial Intelligence*, G.M.P. O'Hare and N.R. Jennings, eds. 1996, John Wiley and Sons: London, p. 169-185, ISBN 0-4710-0675-0.

84. Hastings, T.E. and A.S.M. Sajejev, *A vector-based approach to software size measurement and effort estimation*. IEEE Transactions on Software Engineering, 2001. **27**(4): p. 337-350.
85. Hilaire, V., et al., *Formal Specification and Prototyping of Multi-Agent Systems*, in *Engineering Societies in the Agents World, First International Workshop, ESAW 2000, Berlin, Germany, August 21, 2000*, A. Omicini, R. Tolksdorf, and F. Zambonelli, eds. 2001, Springer Verlag: Berlin, p. 114-127, ISBN 3-5404-1477-0.
86. Hong, S., G.v.d. Goor, and S. Brinkkemper, *A Formal Approach to the Comparison of Object-Oriented Analysis and Design Methodologies*, in *The Twenty-Sixth Annual Hawaii International Conference on System Sciences*. 1993, IEEE Press: Hawaii. p. 689-699.
87. Huhns, M.N. and L.M. Stephens, *Multiagent Systems and Societies of Agents*, in *Multi-Agent Systems: a Modern Approach to Distributed Artificial Intelligence*, G. Weiss, ed. 1999, MIT Press: New York, p. 79-120, ISBN 0-2627-3131-2.
88. Hullermeier, E. and C. Zimmermann, *A Two-Phase Search Method for Solving Configuration Problems*. 1998, Dept. of Mathematics and Computer Science, University of Paderborn: Paderborn, Germany. p. 42.
89. IDC, *Strengthening End-to-End eBusiness Security and Privacy*, in *Information Security Services Worldwide Market Forecast and Analysis, 1999-2004*,. 2000, IDC Rep No. 23166. p. 8.
90. Iglesias, C.A., M. Garrijo, and J.C. Gonzalez, *A Survey of Agent-Oriented Methodologies*, in *Proceedings of the 5th International Workshop on Intelligent Agents {V}: Agent Theories, Architectures, and Languages (ATAL-98)*, J. Muller, M.P. Singh, and A.S. Rao, eds. 1999, Springer-Verlag: Heidelberg, Germany, p. 317-330, ISBN 3-540-65713-4.
91. Iglesias, C.A., et al., *Analysis and Design of Multiagent Systems using MAS-CommonKADS*, in *Intelligent Agents IV: Agent Theories, Architectures, and Languages (ATAL '97)*, M.P. Singh, A.S. Rao, and M.J. Wooldridge, eds. 1998, Springer Verlag: Berlin, Germany, p. 313-326, ISBN 3-540-64162-9.
92. Institute of Electrical and Electronics Engineers, *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries*. 1990.
93. Ishida, T., L. Gasser, and M. Yokoo, *Organisation Self-Design in Distributed Production Systems*. IEEE Transactions on Knowledge and Data Engineering, 1992. **4**(2): p. 123-134.
94. Jacobson, I., M. Ericsson, and A. Jacobson, *The Object Advantage: Business Process Re-engineering with Object Technology*. 1994, Menlo Park, CA: Addison Wesley Publishing Company, p. 347, ISBN 0-201-42289-1.
95. JavaCC, *Java Compiler Compiler*. 2000: <http://www.suntest.com>.
96. Jennings, N.R., *On Agent-based Software Engineering*. Artificial Intelligence, 2000. **117**: p. 277-296.
97. Jennings, N.R., *An agent-based approach for building complex software systems*. Communications of the ACM, 2001. **44**(4): p. 35-41.
98. Jennings, N.R. and M. Wooldridge, *Agent-Oriented Software Engineering*, in *Handbook of Agent Technology*, J. Bradshaw, ed. (to appear), AAAI/MIT Press.
99. Jennings, N.R., et al., *Autonomous Agents for Business Process Management*. Applied Artificial Intelligence, 2000. **14**(2): p. 145-189.

100. Jennings, N.R. and M.J. Wooldridge, eds. *Agent Technology: Foundations, Applications and Markets*. Software Agents. 1998, Springer Verlag: Berlin, Germany, p. 352, ISBN 3-5406-3591-2.
101. Karageorgos, A. and S. Thompson, *Multi-Agent System Design Using Role Models*, Patent Application, *BT Case Ref A26117*. 2001: UK.
102. Karageorgos, A., S. Thompson, and N. Mehandjiev. *Semi-Automatic Design of Agent Organisations*. in *Proceedings of the ACM Symposium on Applied Computing (SAC 2002)*. 2002. Madrid: ACM Press.
103. Karageorgos, A., S. Thompson, and N. Mehandjiev, *Agent-Based System Design for B2B E-commerce*. International Journal of Electronic Commerce, Special Issue in Agents in B2B, 2002. 7(1): p. 59-90.
104. Karageorgos, A., N. Mehandjiev, and S. Thompson, *RAMASD: a semi-automatic method for designing agent organisations*. Knowledge Engineering Review, Special Issue on Coordination and Knowledge Engineering, 2002. 17(4): p. 57-84.
105. Kauffman, R.J., M. Subramani, and C.A. Wood. *Analysing Information Intermediaries in Electronic Brokerage*. in *Proceedings of the 33rd Hawaii International Conference on System Sciences*. 2000. Hawaii: IEEE Computer Society Press.
106. Kendall, E.A. *Agent Roles and Role Models: New Abstractions for Multiagent System Analysis and Design*. in *AIP'98, Intelligent Agents for Information and Process Management, German Conference on Artificial Intelligence*. 1998. Bremen, Germany.
107. Kendall, E.A. *Role Modelling for Agent System Analysis, Design and Implementation*. in *First International Symposium on Agent Systems and Applications Third International Symposium on Mobile Agents*. 1999. Palm Springs, California.
108. Kendall, E.A., *Role models - patterns of agent system analysis and design*. BT Technology Journal, 1999. 17(4): p. 46-57.
109. Kendall, E.A., *Agent Analysis and Design with Role Models*, in *Volume 1: Overview*. 1999, BT Exact Technologies: Martlesham Heath, UK. p. 89.
110. Kendall, E.A., *Agent Analysis and Design with Role Models*, in *Volume 2: Role Models for Agent Enhanced Workflow and Business Process Management*. 1999, BT Exact Technologies: Martlesham Heath, UK. p. 57.
111. Kendall, E.A., *Agent Software Engineering with Role Modelling*, in *Agent-Oriented Software Engineering II, First International Workshop (AOSE 2000), Limerick, Ireland*, P. Ciancarini and M.J. Wooldridge, eds. 2001, Springer Verlag: Berlin, p. 163-169, ISBN 3-540-41594-7.
112. Kendall, E.A. and L. Zhao. *Capturing and Structuring Goals*. in *Workshop on Use Case Patterns, Object Oriented Programming Systems Languages and Architectures (OOPSLA)*,. 1998. Vancouver, British Columbia, Canada: ACM Press.
113. Kinny, D. and M. Georgeff, *Modelling and Design of Multi-Agent Systems*, in *Intelligent Agents {III}: Agent Theories, Architectures, and Languages*, J.P. Muller, M.J. Wooldridge, and N.R. Jennings, eds. 1997, Springer Verlag: Berlin, p. 1-20, ISBN 3-5406-2507-0.
114. Klein, M. and R. Kazman, *Attribute-Based Architectural Styles*. 1999, Software Engineering Institute: Pittsburgh, PA 15213-3890. p. 41.
115. Koubarakis, M. and D. Plexoudakis, *Business process modelling and Design: a formal model and methodology*. BT Technology Journal, 1999. 17(4): p. 23-35.
116. Kristensen, B.B. *Object-Oriented Modelling with Roles*. in *4th International Conference on Object-Oriented Information Systems*. 1997. Brisbane, Australia.

117. Kristoffersen, S. and F. Ljungberg, *MobiCom: Networking Dispersed Groups*. Interacting with Computers, 1998. **10**: p. 55-65.
118. Kruchten, P.B., *The 4+1 View Model of Architecture*. IEEE Software, 1995. **12**(6): p. 5-33.
119. Lee, L., et al., *The stability, scalability and performance of multi-agent systems*. BT Technology Journal, 1998. **16**(3): p. 94-103.
120. Lee, M.T.-C., et al., *Domain-Specific High-Level Modeling and Synthesis for ATM Switch Prototyping*. Design Automation for Embedded Systems, 1997. **2**(3/4): p. 319-338.
121. Lesaint, D., et al., *Engineering Dynamic Scheduler for Work Manager*. BT Technology Journal, 1998. **16**(3).
122. Lind, J., *MASSIVE: Software Engineering for Multi-Agent Systems*, PhD Thesis. Dept of Computer Science. 2000, DFKI: Saarbrücken.
123. Loukopoulos, P. and V. Karakostas, *System Requirements Engineering*. 1995, London: McGraw-Hill, p. 160, ISBN 0-07-707843-8.
124. Lowry, M.R. and R.D. McCartney, eds. *Automating Software Design*. 1991, AAAI Press: Menlo Park, CA, p. 662, ISBN 0-262-62080-4.
125. Ludwig, H., et al., *A Service Level Agreement Language for Dynamic Electronic Services*. 2002, Thomas J. Watson Research Center: New York. p. 14.
126. Lupu, E., Z. Milosevic, and M. Sloman. *Use of Roles and Policies for Specifying and Managing a Virtual Enterprise*. in *RIDE*. 1999.
127. Lupu, E.C., *A Role Based Framework for Distributed Systems Management*, PhD Thesis, *Department of Computing*. 1998, Imperial College of Science, Technology and Medicine: London.
128. MacDonell, S.G., *Comparative review of functional complexity assessment methods for effort estimation*. Software Engineering Journal, 1994. **9**(3): p. 107-116.
129. MacDonell, S.G., *Determining delivered functional error content based on the complexity of CASE specifications*. New Zealand Journal of Computing, 1994a. **5**(1): p. 57-65.
130. MacDonell, S.G., *Establishing relationships between specification size and software process effort in CASE environments*. Information and Software Technology, 1997. **39**(1): p. 35-45.
131. Madsen, J., et al., *LYCOS: the Lyngby Co-Synthesis System*. Design Automation for Embedded Systems, 1997. **2**(2): p. 195-235.
132. Maher, M.L., *Process models of design synthesis*, in *AI Magazine*. 1990. p. 49-58.
133. Mahling, D.e. and R.C. King, *A Goal-based Workflow System for Multiagent task coordination*. Journal of Organisational Computing and Electronic Commerce, 1999. **9**(1): p. 57-82.
134. Maimon, O. and D. Braha, *On the Complexity of the Design Synthesis Problem*. IEEE Transactions on Systems, Man, And Cybernetics-Part A: Systems and Humans, 1996. **26**(1).
135. Maisano, P., *JACK Intelligent Agents™ User Guide*. 2002, Agent Oriented Software Pty. Ltd.: Melbourne, Australia.
136. Malville, E. and F. Bourdon. *Task Allocation: A group self Design Approach*. in *International Conference on Multi-Agent Systems*. 1998. Paris: IEEE Press.

137. Mattsson, M.M., *A Comparative Study of Three New OO Methods*. 1995, Department of Computer Science, University of Karlskrona/Ronneby: Ronneby. p. 31.
138. McConnell, M. and B.A. Hamilton, *Information Assurance in the Twenty-First Century*, in *IEEE Computer, supplement on Security and Privacy*. 2002. p. 16-19.
139. Metzger, A. and S. Quells, *A Reuse- and Prototyping-based Approach for the Specification of Building Automation Systems*, in *OMER-2 Workshop Proceedings*, A. Schuerr, ed. 2001, University of the Federal Armed Forces, Germany: Munich. p. 3-9.
140. Morgenthal, J.P., *Which B2B Exchange Is Right for You?*, in *Software Magazine*. 2001.
141. Nada, N. and D.C. Rine, *Three empirical evaluations of a software reuse reference model*. *Annals of Software Engineering*, 2000. **10**(1-4): p. 225-259.
142. Newcomb, T.M. et al. *Social Psychology*. 1966, London: Routledge & Kegan Paul, ISBN 0-7100-1890-8.
143. Ng, K., J. Kramer, and J. Magee, *A CASE Tool for Software Architecture Design*. *Automated Software Engineering*, 1996. **3**(3/4): p. 261-284.
144. Niemann, R. and P. Marwedel, *An Algorithm for Hardware/Software Partitioning Using Mixed Integer Linear Programming*. *Design Automation for Embedded Systems*, 1997. **2**(2): p. 165-193.
145. Nixon, B.A., *Management of Performance Requirements for Information Systems*. *IEEE Transactions on Software Engineering*, 2000. **26**(12): p. 1122-1146.
146. Nwana, H.S., *Software Agents: An Overview*. *Knowledge Engineering Review*, 1996. **11**(3): p. 205-244.
147. Nwana, H.S., et al., *Zeus: A Toolkit for Building Distributed Multi-Agent Systems*. *Applied Artificial Intelligence Journal*, 1999. **13**(1): p. 129 - 185.
148. Objectspace Inc., *Voyager 2.0 User's Manual*. 1997, Objectspace Inc.
149. OMG, *Unified Modelling Language Specification, ver. 2.0*. 2000.
150. Omicini, A., *SODA : Societies and Infrastructures in the Analysis and Design of Agent-based Systems*, in *Agent-Oriented Software Engineering II, First International Workshop (AOSE 2000)*, Limerick, Ireland, P. Ciancarini and M.J. Wooldridge, Editors. 2001, Springer Verlag: Berlin, p. 185-193, ISBN 3-540-41594-7.
151. Ould, M., *Modelling and Analysis for Reengineering and Improvement*. 1995, West Sussex, UK: John Wiley & Sons, p. 224, ISBN 0-4719-5352-0.
152. Parsons, M.G., D.J. Singer, and J.A. Sauter. *A Hybrid Agent Approach for set-based conceptual ship design*. in *International Conference on Computer Applications in Shipbuilding*. 1999. Cambridge, MA.
153. Parsons, T., *The Social System*. 1964, Glenoe, ILL: The Free Press, ISBN 0-0292-4190-1.
154. Parunak, H.V.D., *A Practitioners' Review of Industrial Agent Applications*. *Autonomous Agents and Multi-Agent Systems*, 2000. **3**(4): p. 389-407.
155. Parunak, H.V.D., et al. *A Marketplace of Design Agents for Distributed Concurrent Set-Based Design*. in *Fourth International Conference on Concurrent Engineering, Research and Applications*. 1997. Troy, Michigan.
156. Parunak, H.V.D., et al. *Distinguishing Environmental and Agent Dynamics: A Case Study in Abstraction and Alternate Modeling Technologies*. in *Engineering Societies in the Agents World, First International Workshop, ESAW 2000, Berlin, Germany, August 21, 2000*, A. Omicini, R. Tolksdorf, and F. Zambonelli, eds. 2001, Springer Verlag: Berlin, p. 19-33, ISBN 3-5404-1477-0.

157. Parunak, V., J. Sauter, and S. Clark, *Toward the Specification and Design of Industrial Synthetic Ecosystems*, in *Intelligent Agents IV: Agent Theories, Architectures, and Languages*, M.P. Singh, A. Rao, and M.J. Wooldridge, Editors. 1998, Springer Verlag: Berlin, p. 45-59, ISBN 3-540-64162-9.
158. Parunak, V.D., et al., *The RAPPID Project: Symbiosis between Industrial Requirements and MAS Research*. Autonomous Agents and Multi-Agent Systems, 1999. 2(2): p. 111-140.
159. Pattison, H.E., D.D. Corkill, and V.R. Lesser, *Instantiating Descriptions of Organisational Structures*, in *Distributed Artificial Intelligence*, M.N. Huhns, ed. 1987, Pitman: London, p. 59-96, ISBN 0-273-08778-9.
160. Petrie, C., *Agent-Based Software Engineering*, in *Agent-Oriented Software Engineering II, First International Workshop (AOSE 2000), Limerick, Ireland*, P. Ciancarini and M.J. Wooldridge, eds. 2000, Springer Verlag: Berlin, p. 58-76, ISBN 3-540-41594-7.
161. Pinzon, L.E., et al., *A Comparative Study of Synthesis Methods for Discrete Event Controllers*. Formal Methods in System Design, 1999. 15(2): p. 99-122.
162. Rao, A.S. and M.P. Georgeff, *BDI-agents: from theory to practice*, in *Proceedings of the First International Conference on Multiagent Systems*. 1995: San Francisco.
163. Reenskaug, T., P. Wold, and O.A. Lehne, *Working with Objects, The OOram Software Engineering Method*. 1996, Greenwich: Manning Publications, p. 420, ISBN 0-1345-2930-8.
164. Renner, F.-M., J. Becker, and M. Glesner, *Communication Performance Models for Architecture-Precise Prototyping of Real-Time Embedded Systems*. Design Automation for Embedded Systems, 2000. 5(3-4): p. 351-363.
165. Reticular Systems Inc, *Agent Construction Tools*. 2002.
166. Richle, D. *Composite Design Patterns*. in *OOPSLA'97 Proceedings of the 1997 Conference on Object-Oriented Programming Systems, Languages and Applications*. 1997: ACM Press.
167. Richle, D., *A Role-Based Design Pattern Catalog of Atomic and Composite Patterns Structured by Pattern Purpose*. 1997, Ubilab, Union Bank of Switzerland: Zurich. p. 48.
168. Richle, D. and T. Gross. *Role model based framework design and integration*. in *OOPSLA'98 Proceedings of the 1998 Conference on Object-Oriented Programming Systems, Languages and Applications*. 1998: ACM Press.
169. Ricordel, P.-M. and Y. Demazeau, *From Analysis to Deployment: a Multi-Agent Platform Survey*, in *Engineering Societies in the Agents World, First International Workshop, ESAW 2000, Berlin, Germany, August 21, 2000*, A. Omicini, R. Tolksdorf, and F. Zambonelli, eds. 2001, Springer Verlag: Berlin, p. 93-105, ISBN 3-5404-1477-0.
170. Roeckle, H., G. Schimpf, and R. Weidinger. *Process-oriented approach for role-finding to implement role-based security administration in a large industrial organization*. in *ACM Workshop on Role-based Access Control*. 2000. Berlin, Germany: ACM Press.
171. Romanosky, S., *Security Design Patterns v 1.4*. 2001, Security Focus: San Francisco, CA. p. 19.
172. Rumbaugh, J. et al., *Object Oriented Modelling and Design*. 1991, New York: Prentice Hall International, p. 528, ISBN 0-13-629841-9.
173. Russel, S. and D. Subramanian, *Provably bounded-optimal agents*. Journal of AI Research, 1995. 2: p. 575-609.
174. Samuel, A. and J. Weir, *Introduction to Engineering Design: Modelling, Synthesis and Problem Solving*. 1999, Oxford: Butterworth-Heinemann, p. 416, ISBN 0-7506-4282-3.

175. Schiefeloe, P.M. and T.G. Syvertsen, *Coordination in Knowledge-Intensive Organisations*, in *Coordination Technology for Collaborative Applications: Organizations, Processes and Agents*, W. Conen and G. Neumann, eds. 1999, Springer Verlag, p. 9-23, ISBN 3-5406-4170-X.
176. Schreiber, G., et al., *Knowledge Engineering and Management: The CommonKADS Methodology*. 2000, New York: MIT Press, p. 455.
177. Scott, W.R., *Organisations: Rational, Natural and Open Systems*. 2003, New York, NY: Prentice Hall International, p. 430, ISBN 0-13-016559-X.
178. Serugendo, G.D.M. and C.V. Aart, *Engineering Self-Organising Applications Working Group*. 2002, Agentcities.NET Project.
179. Shaw, M. and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*. 1996, New York, NY: Prentice Hall Publishing, p. 242, ISBN 0-13-182957-2.
180. Shehory, O., S. Kraus, and O. Yadgar, *Emergent Cooperative Goal-Satisfaction in Large Scale Automated-Agent Systems*. Artificial Intelligence, 1999. **110**(1): p. 1-55.
181. Shen, W. and D.H. Norrie, *Agent-Based Systems for Intelligent Manufacturing: A State-of-the-Art Survey*. Knowledge and Information Systems, an International Journal, 1999. **1**(2): p. 129-156.
182. Silva, A.R., et al., *Towards a Reference Model for Surveying Mobile Agent Systems*. Autonomous Agents and Multi-Agent Systems, 2001. **4**(3): p. 187-231.
183. Simonyi, C., *Intentional Programming - Innovation in the Legacy Age*. 1996, Microsoft Corporation: IFIP WG 2.1 meeting., p. 25.
184. Smith, R.G., *The Contract Net Protocol. High Level Communication and Control in a Distributed Problem Solver*. IEEE Transactions on Computers, 1980. **29**(12): p. 1104-1113.
185. So, Y.-p. and E.H. Durfee, *Designing Organisations for Computational Agents*, in *Simulating Organisations: Computational Models of institutions and groups*, M.J. Prietula, K.M. Carley, and L. Gasser, eds. 1998, AAAI Press: Menlo Park, CA, p. 47-64, ISBN 0-262-66108-X.
186. Sparkman, C.H., S.A. DeLoach, and A.L. Self, *Automated Derivation of Complex Agent Architectures from Analysis Specifications*, in *Agent-Oriented Software Engineering II, Second International Workshop (AOSE 2001), Montreal, Canada*, M.J. Wooldridge, G. Weis, and P. Ciancarini, eds. 2001, Springer Verlag: Berlin, p. 278-296, ISBN 3-540-43282-5.
187. Stark, J., et al., *ACSOSS: a case study applying the MESSAGE analysis method*. 2001, BT Exact Technologies: Martlesham Heath.
188. Steimann, F., *On the representation of roles in object-oriented and conceptual modelling*. Data and Knowledge Engineering, 2000. **35**: p. 83-106.
189. Stergiou, K., *Representation and Reasoning with Non-Binary Constraints*, PhD Thesis, Department of Computer Science. 2001, University of Strathclyde: Glasgow, UK. p. 174.
190. Szymanek, R., F. Gruian, and K. Kuchcinski. *Digital Systems Design Using Constraint Logic Programming*. in *Practical Application of Constraint Logic Programming (PACLP'2000)*, 2000. Manchester: Practical Applications Company.
191. Tambe, M., *Towards flexible teamwork*. Journal of AI Research, 1997. **7**: p. 83-124.
192. Tambe, M., D.V. Pynadath, and N. Chauvat, *Building Dynamic Agent Organisations in Cyberspace*. IEEE Internet Computing, 2000. **4**(2): p. 65-73.

193. Tambe, M., D.V. Pynadath, and N. Chauvat. *Rapid integration and coordination of heterogeneous distributed agents for collaborative enterprises*. in *DARPA JFACC symposium on advances in Enterprise Control*. 2000.
194. Tekinerdogan, B., *Synthesis-based Software Design*, PhD Thesis, *Department of Computer Science*. 2000, University of Twente: Twente. p. 226.
195. The Object Agency Inc, *A Comparison of Object-Oriented Development Methodologies*. 1995, The Object Agency, Inc.
196. Thomas, E.J. and B.J. Biddle, *The Nature and History of Role Theory*, in *Role Theory: Concepts and Research*, B.J. Biddle and E.J. Thomas, eds. 1979, John Wiley Publishing: London, p. 3-19, ISBN 0-471-07215-X.
197. Thompson, S., *The Zeus mailing list*, (zeus@jiscmail.ac.uk). 1999, BT Exact Technologies.
198. Thompson, S., *ZEUS Methodology Documentation Part V: FIPA & Zeus*. 2001, BT Exact Technologies.
199. Thompson, S.G. and B.R. Odgers. *Collaborative Personal Agents for Team Working*. in *Proceedings of 2000 Artificial Intelligence and Simulation of Behaviour (AISB) Symposium*. 2000. Birmingham, England, ISBN 1-902956-14-6.
200. Wagner, G., *Agent-Oriented Analysis and Design of Organizational Information Systems*, in *Proc. of Fourth IEEE International Baltic Workshop on Databases and Information Systems*. 2000: Vilnius (Lithuania).
201. Weiss, G., ed. *Multi-Agent Systems: a Modern Approach to Distributed Artificial Intelligence*. 1999, MIT Press: New York, p. 619, ISBN 0-262-23203-0.
202. Werner, E., *Cooperating Agents: A Unified Theory of Communication and Social Structure*, in *Distributed Artificial Intelligence*, L. Gasser and M.N. Huhns, eds. 1989, Pitman/Morgan Kaufmann: London, p. 3-36, ISBN 1-558-60092-2.
203. Willmott, S.N., et al., *Agentcities: A Worldwide Open Agent Network*. Agentlink News, 2001(8): p. 13-15.
204. Wong, R.K. *RBAC support in object-oriented role databases*. in *ACM Workshop on Role-based Access Control*. 1997. Fairfax, VA USA: ACM Press.
205. Wood, M. and S.A. DeLoach, *An Overview of the Multiagent Systems Engineering Methodology*, in *Agent-Oriented Software Engineering I, First International Workshop (AOSE 2000), Limerick, Ireland*, P. Ciancarini and M.J. Wooldridge, eds. 2001, Springer Verlag: Berlin, p. 207-221, ISBN 3-540-41594-7.
206. Wooldridge, M., *Intelligent Agents*, in *Multi-Agent Systems: a Modern Approach to Distributed Artificial Intelligence*, G. Weiss, ed. 1999, MIT Press: New York, p. 27-77, ISBN 0-262-23203-0.
207. Wooldridge, M., *On the Sources of Complexity in Agent Design*. Applied Artificial Intelligence, 2000. **14**(7): p. 623-644.
208. Wooldridge, M. and P. Ciancarini, *Agent-Oriented Software Engineering: The State of the Art*, in *Agent-Oriented Software Engineering I, First International Workshop (AOSE 2000), Limerick, Ireland*, P. Ciancarini and M.J. Wooldridge, eds. 2001, Springer-Verlag: Berlin, p. 1-28, ISBN 3-540-41594-7.
209. Wooldridge, M., N.R. Jennings, and D. Kinny, *The Gaia methodology for agent-oriented analysis and design*. International Journal of Autonomous Agents and Multi-Agent Systems, 2000. **3**(3): p. 285-312.
210. Wooldridge, M.J. and N.R. Jennings, *Intelligent Agents: Theory and Practice*. Knowledge Engineering Review, 1995. **10**(2): p. 115-152.

211. Wooldridge, M.J. and N.R. Jennings, *Agent Based Software Engineering*. IEE Proceedings in Software Engineering, 1997. **144**(1): p. 26-37.
212. Wooldridge, M.J. and N.R. Jennings, *Software Engineering with Agents: Pitfalls and Pratfalls*. IEEE Internet Computing, 1999. **3**(3): p. 20-27.
213. Wyns, J., *Reference architecture for Holonic Manufacturing Systems - the key to support evolution and reconfiguration*, PhD Thesis, Department of Mechanical Engineering. 1999, KU Leuven: Leuven. p. 274.
214. Yacoub, S.M. and H.H. Ammar. *Pattern-Oriented Analysis and Design (POAD): A Structural Composition Approach to Glue Design Patterns*. in *Proceedings of the : Technology of Object-Oriented Languages and Systems (TOOLS-34"00)*. 2000: IEEE Press.
215. Yacoub, S.M., H. Xue, and H.H. Ammar. *POD: A Composition Environment for Pattern-Oriented Design*. in *Proceedings of the : Technology of Object-Oriented Languages and Systems (TOOLS-34"00)*. 2000: IEEE Press.
216. Yoder, J. and J. Barcalow[*Architectural Patterns for Enabling Application Security*, in *The 4th Pattern Languages of Programming Conference*. 1997, Washington University: Allerton Park, Monticello, Illinois, USA.
217. Yu, E.S.K. *Models for Supporting the Redesign of Organisational Work*. in *ACM Conference on Organisational Computing (COOCS'95)*. 1995. Milpitas California: ACM Press.
218. Yu, E.S.K. and J. Mylopoulos. *An Actor Dependency Model of Organisational Work with application to Business Process Reengineering*. in *ACM Conference on Organisational Computing (COOCS'93)*. 1993. Milpitas California: ACM Press.
219. Yu, L. and B.F. Schmid. *A Conceptual Framework for Agent Oriented and Role Based Workflow Modelling*. in *CaiSE Workshop Conference on Agent Oriented Information Systems (AOIS'99)*. 1999. Heidelberg: MIT Press.
220. Zambonelli, F., N.R. Jennings, and M. Wooldridge, *Organisational Abstractions for the Analysis and Design of Multi-Agent Systems*, in *Agent-Oriented Software Engineering II, First International Workshop (AOSE 2000), Limerick, Ireland*, P. Ciancarini and M.J. Wooldridge, eds. 2001, Springer Verlag: Berlin, p. 235-250, ISBN 3-540-41594-7.
221. Zambonelli, F., N.R. Jennings, and M. Wooldridge, *Organisational Rules as an Abstraction for the Analysis and Design of Multi-Agent Systems*. International Journal of Software Engineering and Knowledge Engineering, 2001. **11**(3): p. 303-328.
222. Zambonelli, F., et al., *Agent-Oriented Software Engineering for Internet Applications*, in *Coordination of Internet Agents: Models, Technologies and Applications*, A. Omicini, et al., eds. 2001, Springer-Verlag: Berlin Heidelberg, p. 326-346, ISBN 3-5404-1613-7.
223. Zelkowitz, M.V. and D.R. Wallace, *Experimental Models for Validating Technology*, in *IEEE Computer*. 1998. p. 23-31.
224. Zhao, L. and E. Kendall. *Role modelling for component design*. in *TOOLS 33, Technology of Object-Oriented Languages and Systems*. 2000: IEEE Computer Society, ISBN 0-7695-0731-x.
225. Zini, F., *Caselp, A Rapid Prototyping Environment For Agent Based Software*. 2001, ITC IRST: Trento, Italy. p. 163.
226. Zuse, H., *Software Complexity - Measures and Methods*. first ed. 1991, Berlin, New York: Verlag de Gruyter, p. 605 ISBN 3-11-0122260-X.

Table of Contents

CHAPTER 1..... 1

INTRODUCTION1

1.1	MOTIVATION	1
1.2	CONTEXT OF THE THESIS	2
1.3	ISSUES AND CHALLENGES.....	3
1.3.1	<i>The ABS Design Complexity Problem.....</i>	<i>3</i>
1.3.2	<i>Reusing Design Knowledge.....</i>	<i>4</i>
1.3.3	<i>Non-Functional Aspects and Design Heuristics</i>	<i>4</i>
1.4	AIMS AND OBJECTIVES	5
1.5	MAIN CONTRIBUTIONS	6
1.6	RESEARCH METHODOLOGY	7
1.7	CASE STUDY DESCRIPTIONS	8
1.8	THESIS ORGANISATION.....	9

CHAPTER 2..... 11

AGENT-BASED SYSTEM DESIGN11

2.1	DESIGNING ABSS	11
2.2	OVERVIEW OF AGENT CONCEPTS.....	11
2.2.1	<i>Agent-Oriented vs Object-Oriented Approaches</i>	<i>11</i>
2.2.2	<i>Defining the Term ‘Agent’.....</i>	<i>13</i>
2.2.3	<i>A Simple Agent Formal Model</i>	<i>15</i>
2.2.4	<i>An Example of a Simple Agent.....</i>	<i>15</i>
2.2.5	<i>Using Roles to Model Agent Behaviour</i>	<i>16</i>
2.2.6	<i>Agent Architecture</i>	<i>17</i>
2.3	AGENT-BASED SYSTEMS	19
2.3.1	<i>Overview</i>	<i>19</i>
2.3.2	<i>Interaction in ABSs</i>	<i>20</i>
2.3.3	<i>ABSs as Organisations of Agents.....</i>	<i>20</i>
2.3.4	<i>Software Complexity and ABS Design</i>	<i>21</i>
2.3.4.1	Complexity in Software Engineering.....	21
2.3.4.2	Complexity in ABS Design.....	22
2.4	CLASSIFICATION OF ABS ENGINEERING APPROACHES.....	24

2.4.1	<i>Ad-hoc Approaches</i>	25
2.4.2	<i>Formal Approaches</i>	25
2.4.3	<i>Informal Approaches</i>	26
2.4.3.1	Approaches Based on Object-Oriented Software Engineering	27
2.4.3.2	Approaches Based on Information Systems Engineering.....	27
2.4.3.3	Approaches Based on Knowledge Engineering.....	28
2.4.3.4	Tool-Based Approaches.....	28
2.4.4	<i>Dynamic Approaches</i>	29
2.4.5	<i>Overall Assessment</i>	29
2.5	SUMMARY	31

CHAPTER 3..... 33

ASSESSMENT OF ABS ENGINEERING APPROACHES33

3.1	AN EVALUATION FRAMEWORK FOR ABS DESIGN.....	33
3.1.1	<i>Concepts</i>	34
3.1.2	<i>Models</i>	35
3.1.3	<i>Process</i>	36
3.1.4	<i>Pragmatics</i>	37
3.2	COMPARATIVE EVALUATION OF ABS ENGINEERING APPROACHES	40
3.3	IMPLICATIONS FOR FURTHER RESEARCH	41
3.3.1	<i>Support for Design Heuristics</i>	42
3.3.2	<i>Organisational Settings</i>	42
3.3.3	<i>Collective Behaviour</i>	43
3.3.4	<i>Non-Functional Aspects</i>	44
3.3.5	<i>Automating the Design Process</i>	45
3.3.6	<i>Working at Different Abstraction Levels</i>	45
3.4	SUMMARY	47

CHAPTER 4..... 49

USING ROLE MODELLING FOR ABS DESIGN49

4.1	COMPLETE ROLE MODELLING APPROACHES.....	49
4.2	MODELLING SOCIAL BEHAVIOUR USING ROLES	50
4.2.1	<i>Defining the Term ‘Role’: a Social View</i>	50
4.2.1.1	Social Aspects of Role Definitions.....	50
4.2.1.2	Role Relationship Zones.....	51
4.2.2	<i>Overview of Role Theory</i>	52

4.2.3	<i>Role Theoretic Concepts</i>	53
4.2.4	<i>Role Dependency Relations</i>	55
4.2.5	<i>Role Identification and Role Types</i>	56
4.3	USING ROLES IN INFORMATION SYSTEMS MODELLING	57
4.3.1	<i>Roles in Business Systems Modelling</i>	57
4.3.2	<i>Role-Based Access Control in Distributed Systems Management</i>	59
4.3.3	<i>Roles in Object Oriented Software engineering</i>	60
4.3.3.1	Defining Roles in Object Oriented Software Engineering	61
4.3.3.2	Role Properties	63
4.3.3.3	Role Relationships, Synthesis and Synergy	65
4.3.4	<i>Roles in ABS Modelling</i>	66
4.3.4.1	Modelling Goal-Based Interactions Using Roles	67
4.3.4.2	Modelling Organisational Settings Using Roles	68
4.4	USING ROLES FOR THE DESIGN OF ABSS	69
4.4.1	<i>Comparison of Role Modelling Approaches</i>	69
4.4.2	<i>Formalising Role Dependency Relations</i>	71
4.5	SUMMARY	71

CHAPTER 5..... 73

THE RAMASD METHOD.....73

5.1	USING ROLE MODELLING AND SYNTHESIS FOR ABS DESIGN	73
5.2	ROLE MODELLING IN RAMASD	74
5.2.1	<i>Defining Roles and Role Models</i>	74
5.2.1.1	Role Characteristics	74
5.2.1.2	Properties of Roles and Role Models	76
5.2.2	<i>Representing and Using Role Models</i>	77
5.2.3	<i>Role Model Types</i>	77
5.2.4	<i>Identification of Roles in the Application Domain</i>	78
5.2.4.1	Criteria for Role Identification	78
5.2.4.2	Goal-Oriented Role Identification	79
5.2.4.3	Role Identification for an e-Business Security System	82
5.2.5	<i>Management of the Role Modelling Process</i>	84
5.2.6	<i>Consistency of Role-Based Specifications</i>	85
5.2.7	<i>Rigorous Role Assignment Using Role Algebra</i>	85
5.2.7.1	Relations in the Role Algebra	86
5.2.7.2	Semantics of the Role Algebra	88
5.2.7.3	Graphical Representation of Role Relations	90
5.3	APPLYING THE SYNTHESIS CONCEPT TO ABS DESIGN	90

5.3.1	<i>Synthesis in Traditional Engineering</i>	90
5.3.2	<i>A Synthesis-Based Design Process Model</i>	92
5.4	THE RAMASD DESIGN PROCESS	95
5.5	THE INNOVATIVE FEATURES OF RAMASD.....	97
5.5.1	<i>The Philosophy of the RAMASD Approach</i>	97
5.5.2	<i>Reusing Collective Behaviour</i>	97
5.5.2.1	Representing and Using Patterns of Behaviour	97
5.5.2.2	An Example of Behaviour Reuse	98
5.5.3	<i>Representing Organisational Settings</i>	100
5.5.4	<i>Considering Non-Functional Aspects</i>	102
5.5.5	<i>Considering Design Heuristics</i>	105
5.6	USING RAMASD WITH EXISTING METHODOLOGIES.....	107
5.7	SUMMARY	108

CHAPTER 6.....109

IMPLEMENTATION OF RAMASD.....109

6.1	EXTENDING ZEUS TO SUPPORT RAMASD	109
6.2	THE ZEUS AGENT BUILDING TOOLKIT	111
6.3	EXTENDING ZEUS TO SUPPORT RAMASD	111
6.3.1	<i>The Extended Zeus Agent Development Methodology</i>	111
6.3.2	<i>The Extended Zeus Visual Agent Creator Component</i>	113
6.4	THE RCL CONSTRAINT LANGUAGE	117
6.5	ALLOCATING ROLES TO AGENTS	118
6.6	SUMMARY – CONCLUSIONS	120

CHAPTER 7.....121

CASE STUDIES: MOBILE WORKFORCE SUPPORT AND COVISINT.....121

7.1	APPLYING RAMASD TO REAL WORLD CASES	121
7.2	MOBILE WORKFORCE SUPPORT	122
7.2.1	<i>The Mobile Workforce Support Problem</i>	122
7.2.2	<i>Role Identification</i>	123
7.2.3	<i>Specifying Design Constraints</i>	126
7.2.4	<i>Design Results</i>	127
7.3	EXAMPLE: AN AUTOMOTIVE INDUSTRY B2B EXCHANGE	129
7.3.1	<i>Case Study Overview</i>	129
7.3.2	<i>Role Identification</i>	132

7.3.3	<i>Qualitative Modelling of Non-Functional Aspects</i>	135
7.3.3.1	Security Issues	135
7.3.3.2	Privacy Issues	136
7.3.4	<i>Organisational Settings</i>	137
7.3.5	<i>Role Composition</i>	138
7.3.6	<i>Specifying Design Constraints</i>	139
7.3.7	<i>Role Allocation Results</i>	141
7.4	SUMMARY – CONCLUSIONS	143

CHAPTER 8.....145

EVALUATION OF RAMASD145

8.1	SELECTING AN EVALUATION APPROACH	145
8.1.1	<i>Approaches to Evaluating Software Engineering Methods</i>	145
8.1.2	<i>Evaluating RAMASD</i>	146
8.1.3	<i>Selecting Case Studies and Test Scenarios</i>	147
8.2	FRAMEWORK-BASED EVALUATION	147
8.2.1	<i>Main Features of RAMASD</i>	147
8.2.2	<i>Comparing RAMASD With Other Methods</i>	149
8.3	COMPARISON OF RAMASD AND GAIA	151
8.3.1	<i>Overview of Gaia</i>	151
8.3.2	<i>Applying Gaia in the Mobile Workforce Case Study</i>	152
8.3.3	<i>Limitations of Gaia</i>	155
8.4	DISCUSSION.....	158
8.4.1	<i>Real World Applicability of RAMASD</i>	158
8.4.1.1	The Generality of RAMASD	158
8.4.1.2	The Scalability of RAMASD	159
8.4.2	<i>Novel Aspects of RAMASD</i>	159
8.4.2.1	The Innovative Features of RAMASD.....	159
8.4.2.2	The Role Algebra	160
8.5	SUMMARY	161

CHAPTER 9.....163

CONCLUSIONS163

9.1	REVISITING THE RESEARCH HYPOTHESIS.....	163
9.2	ASSESSING THE THESIS CONTRIBUTIONS	164
9.3	LIMITATIONS OF RAMASD	165

9.4	FURTHER WORK.....	166
9.5	CONCLUDING REMARKS	167
APPENDICES.....		169
APPENDIX A EVALUATION OF ABS DESIGN APPROACHES		169
A.1	RAPPID	169
A.1.1	Overview of RAPPID	169
A.1.2	Evaluation of RAPPID	170
A.1.3	Strengths and Weaknesses of RAPPID.....	171
A.2	DESIRE	172
A.2.1	Overview of DESIRE.....	172
A.2.2	Evaluation of DESIRE.....	174
A.2.3	Strengths and Weaknesses of DESIRE.....	174
A.3	GAIA	175
A.3.1	Overview of Gaia.....	175
A.3.2	Evaluation of Gaia.....	176
A.3.3	Strengths and Weaknesses of Gaia	177
A.4	TROPOS	177
A.4.1	Overview of Tropos.....	177
A.4.2	Evaluation of Tropos.....	178
A.4.3	Strengths and Weaknesses of Tropos	179
A.5	MESSAGE.....	179
A.5.1	Overview of MESSAGE/UML.....	179
A.5.2	Evaluation of MESSAGE.....	182
A.5.3	Strengths and Weaknesses of MESSAGE/UML	183
A.6	ZEUS.....	183
A.6.1	Overview of Zeus Agent Development Methodology	183
A.6.2	Evaluation of Zeus Agent Development Methodology	185
A.6.3	Strengths and Weaknesses of Zeus.....	186
A.7	KARMA/TEAMCORE.....	187
A.7.1	Overview of KARMA/TEAMCORE.....	187
A.7.2	Evaluation of KARMA/TEAMCORE.....	189
A.7.3	Strengths and Weaknesses of KARMA/TEAMCORE	189
APPENDIX B THE ZEUS TOOLKIT.....		191
B.1	THE COMPONENTS OF THE ZEUS TOOLKIT	191
B.1.1	The Agent Component Library	191

<i>B.1.2</i>	<i>The Visualisation Tools.....</i>	192
<i>B.1.3</i>	<i>The Agent Building Tools.....</i>	192
B.2	THE ZEUS AGENT SYSTEM REALISATION PROCESS	195
B.3	THE ZEUS UTILITY AGENTS	198
B.4	THE GENERIC ZEUS AGENT.....	200
APPENDIX C RCL EBNF SYNTAX.....		203
REFERENCES		207

Table of Figures

Figure 1.1: PhD research question and solution approach	6
Figure 1.2: Thesis organisation.....	9
Figure 2.1: Perceive-Reason-Act cycle.....	13
Figure 2.2: A simple agent formal model.....	14
Figure 2.3: A container terminal yard agent.....	15
Figure 2.4: Agent internal components	18
Figure 2.5: An agent organisation.....	20
Figure 2.6: Classification of ABS engineering approaches.....	24
Figure 3.1: A framework for comparing ABS engineering approaches with respect to design ..	34
Figure 4.1: Role relationship zones	52
Figure 4.2: Agent-Position-Role dependencies in the Actor-Dependency model	58
Figure 4.3: Role characteristics for distributed systems access control	60
Figure 4.4: Roles as association names	61
Figure 4.5: Roles as patterns of behaviour	61
Figure 4.6: Object–Role relationships (Wong 1997)	63
Figure 4.7: The Bureaucracy pattern represented as a role model (Richle 1997).....	64
Figure 4.8: Sample RRC card for the Bureaucracy pattern (Kendal 1999).....	65
Figure 4.9: A high level view of the supply chain management role model (Kendal 1999).....	66
Figure 4.10: An example MASE role model (DeLoach et., al. 2001).....	67
Figure 5.1: Schematic representation of a role model using UML notation.....	77
Figure 5.2: The phases of a goal-oriented role identification method.....	80
Figure 5.3: Goal cases for an e-business security protection system	82
Figure 5.4: Goal hierarchy tree and role identification for an e-business security system.....	83
Figure 5.5: Identified roles for the e-business security system.....	84
Figure 5.6: Semantics of the role algebra.....	88
Figure 5.7: Graphical notation for the relations of the role algebra	90
Figure 5.8: The synthesis problem solving process	91
Figure 5.9: A generic synthesis-based design process model.....	93
Figure 5.10: Schematic representation of the RAMASD design process.....	95
Figure 5.11: An example of collective behaviour reuse in RAMASD	98
Figure 5.12: RAMASD roles for the conference management system example	99
Figure 5.13: Enforcing organisational rules by appropriate merging of roles	101
Figure 5.14: An example of modelling organisational rules using roles	102
Figure 5.15: Extended actor diagram for an e-cultural system (aft Giorgini et al., 2001)	103
Figure 5.16: Using the FIPA directory facilitator role model for e-culture service brokering ..	104

Figure 5.17: A personal assistant role model	105
Figure 5.18: Spheres of responsibility (Collins et al. 1999).....	106
Figure 5.19: Specifying the spheres of responsibility heuristic using role relations.....	107
Figure 6.1: Conceptual view of the extended Zeus ABS design tool	110
Figure 6.2: The extended Zeus agent development methodology	112
Figure 6.3: The extended Zeus Agent Generator component.....	113
Figure 6.4: The role model and role definition editors	114
Figure 6.5: The Role Constraints Editor component	115
Figure 6.6: The Role Allocation component	116
Figure 6.7: Parts of an RCL specification	118
Figure 6.8: A simple search algorithm for allocating roles to agent types	119
Figure 7.1: A high level view of the mobile workforce coordination case study	122
Figure 7.2: Use case goals for the telephone repair service teams case study.....	124
Figure 7.3: Role models for the telephone repair service teams case study	125
Figure 7.4: Compositional constraints for the telephone repair service teams case study	126
Figure 7.5: Snapshot of the extended Zeus toolkit for the mobile workforce case study.....	128
Figure 7.6: Agent types for the telephone repair service teams case study	129
Figure 7.7: Use case goals for an automotive industry B2B exchange case study	131
Figure 7.8: Role models for the automotive industry B2B exchange case study	133
Figure 7.9: The mediator pattern	137
Figure 7.10: Updated role models based on the mediator pattern	138
Figure 7.11: Compositional constraints for the B2B exchange case study	140
Figure 7.12: Agent types for the B2B exchange case study	142
Figure 7.13: Snapshot of the extended Zeus toolkit for the B2B exchange case study.....	143
Figure A.1: The RAPPID ABS architecture.....	170
Figure A.2: A generic agent model in DESIRE.....	172
Figure A.3: Relations between Gaia models	175
Figure A.4: Knowledge level concepts in MESSAGE/UML	180
Figure A.5: The Zeus agent development methodology	184
Figure A.6: The Zeus agent architecture.....	185
Figure A.7: The KARMA/TEAMCORE Framework.....	187
Figure B.1: The components of the Zeus agent building toolkit (Collins et al. 1999).....	192
Figure B.2: The Control and Society Tools of the Zeus agent building toolkit.....	193
Figure B.3: The agent definition interface of the Zeus agent building tool	193
Figure B.4: The Ontology editor.....	194
Figure B.5: The Zeus agent realisation process (Nwana et al. 1999).....	195
Figure B.6: The flow of information between an agent and a task.....	197

Figure B.7: The Zeus ABS structure (Thompson 2001)	199
Figure B.8: The generic Zeus agent internal structure (Collins and Ndumu 1999).....	201

List of Tables

Table 3.1: Description and ranking of evaluation framework aspects	39
Table 3.2: Comparison of ABS engineering approaches	40
Table 4.1: Strengths and weaknesses of role modelling approaches	70
Table 5.1: Role characteristics.....	74
Table 8.1: Comparing RAMASD with other ABS design methods	150
Table 8.2: The role schema for the REPAIR_WORKER role	153
Table 8.3: Role schemata for the MANAGER and CUSTOMER_HANDLER roles	154
Table 8.4: Role schemata for the TRAVEL_DEPT and EXPERTISE_KNOWLEDGE roles ..	155
Table A.1: Evaluation of RAPPID.....	171
Table A.2: Evaluation of DESIRE.....	174
Table A.3: Evaluation of Gaia.....	176
Table A.4: Evaluation of Tropos	178
Table A.5: Evaluation of MESSAGE/UML.....	182
Table A.6: Evaluation of the Zeus agent development methodology	186
Table A.7: Evaluation of KARMA/TEAMCORE.....	189