

Towards a Comprehensive Agent-Oriented Software Engineering Methodology

Dissertation



vorgelegt dem Fachbereich Wirtschaftswissenschaften,
der Universität Duisburg-Essen
(Campus Essen)

von Tawfig M. Abdelaziz, geboren in Benghazi-Libya

zur Erlangung des Grades
Doktor der Naturwissenschaften (Dr. rer. nat.)

Gutachter:
Prof. Dr. Rainer Unland
Prof. Dr. Cherif Branki

Tag der mündlichen Prüfung: 17.10.2008

ABSTRACT

Recently, agent systems have proven to be a powerful new approach for designing and developing complex and distributed software systems. The agent area is one of the most dynamic and exciting areas in computer science today, because of the agents ability to impact the lives and work of all of us. Developing multi-agent systems for complex and distributed systems entails a robust methodology to assist developers to develop such systems in appropriate way. In the last ten years, many of agent oriented methodologies have been proposed. Although, these methodologies are based on strong basis they still suffer from a set of shortcomings and they still have the problems of traditional distributed systems as well as the difficulties that arise from flexibility requirements and sophisticated interactions. This thesis proposed a new agent oriented software engineering methodology called: Multi-Agent System Development (MASD) for development of multi-agent systems. The new methodology is provided by a set of guidelines, methods, models, and techniques that facilitate a systematic software development process. The thesis makes the following contributions: The main contribution of this thesis is to build a new methodology for the development of multi-agent systems. It is based upon the previous existing methodologies. It is aimed to develop a complete life-cycle methodology for designing and developing MASs. The new methodology is considered as an attempt to solve some of the problems that existing methodologies suffer from. The new methodology is established based on three fundamental aspects: concepts, models, and process. These three aspects are considered as a foundation for building a solid methodology. The concepts are all the necessary MAS concepts that should be available in order to build the models of the new methodology in a correct manner. The models include modeling techniques, modeling languages, a diagramming notation, and tools that can be used to analysis and design the agent system. The process is a set of steps or phases describe how the new methodology works in detail. The new methodology is built to bridge the gap between design models and existing agent implementation languages. It provides refined design models that can be directly implemented in an available programming language or use a dedicated agent-oriented programming language which provides constructs to implement the high-level design concepts such as Jadex, JADE, JACK, etc. The MASD methodology also uses an important concept called triggers and relies heavily on agent roles. The role concept is considered one of the most important aspects that represent agent behaviour. The trigger concept is also considered as an important aspect that represents agent reactivity. The new methodology captures the social agent aspects by utilizing well-known techniques such as use case maps, which enable developers to identify social aspects from the problem specification. MASD methodology is developed based on the essential software engineering issues such as preciseness, accessibility, expressiveness, domain applicability, modularity, refinement, model derivation, traceability, and clear definitions. The MASD methodology is provided by a plain and understandable development process through the methodology phases. It captures the holistic view of the system components, and commutative aspects, which should be recognized before designing the methodology models. This is achieved by using well-known techniques such as UCMs and UML UCDs.

The resulting methodology was obtained by performing several steps. First, a review study “literature review” of different agent methodologies is carried out to capture their strengths and weaknesses. This review study started with the conceptual framework for MAS to discuss the common terms and concepts that are used in the thesis. The aim is to establish the characteristics of agent-oriented methodologies, and see how these characteristics are suited to develop multi-agent systems. Secondly, a requirement for a novel methodology is presented. These requirements are discussed in detail based on the three categories: concepts, models, and process. Thirdly, the new mature methodology is developed based on existing methodologies. The MASD methodology is composed of four phases: the system requirement phase, analysis phase, design phase and implementation phase. The new methodology covers the whole life cycle of agent system development, from requirement analysis, architecture design, and detailed design to implementation. Fourthly, the methodology is illustrated by a case study on an agent-based car rental system. Finally, a framework for evaluating agent-oriented methodologies is performed. Four methodologies including MASD are evaluated and compared by performing a feature analysis. This is carried out by evaluating the strengths and weaknesses of each participating methodology using a proposed evaluation framework called the Multi-agent System Analysis and Design Framework (**MASADF**). The evaluation framework addresses several major aspects of agent-oriented methodologies, such as: concepts, models and process.

ACKNOWLEDGEMENT

First and foremost, I wish to express my deepest gratitude to my supervisor Prof. Rainer Unland (University of Duisburg-Essen, Germany) for his valuable advice and guidance of this work. He has helped me in shaping the research from day one, pushed me to get through the inevitable research setbacks, and encouraged me to achieve to the best of my ability. Without his support and encouragement, this dissertation would not have happened. Special thanks and gratitude to Dr Mohammed Elammari (University of Garyounis, Libya) for his genuine support, valuable advice and sincere comments which helped me a lot to finish this study. I also want to express my thanks and gratitude to Prof. Cherif Branki (University of West Scotland, UK) for reviewing the thesis and for his valuable comments and criticism during the preparation of the manuscript as well as for revising the English language of the manuscript. The Institute for Computer Science and Business Information Systems (ICB), at the University of Duisburg-Essen, Essen, provided support, including working equipments and research space. I am very grateful to the authority of the University of Duisburg-Essen. I would like to thank many people for their support, encouragement and guidance during my years as a graduate student at Duisburg-Essen University

I also thank other persons such as Dr Faheem Bukhatwa (Griffith College Dublin), Dr. Rafi Mansor, Dr Osman Khan (University of Garyounis, Libya), Dr. Stefan Hanenberg (University of Duisburg-Essen, Germany) and Mr. Dawod Kseibat (University of Bedfordshire, UK) for the valuable discussions and comments regarding my research.

I also thank my committee members, whose comments were helpful in refining the dissertation into its final form. Special thanks are due to the staff of the Institute for Computer Science and Business Information Systems (ICB) at the University of Duisburg-Essen, and particularly to Raed Issa, Dominik Stein, Frank Busher, Frau Veronika Muntoni, Gottfried Merkel and Cristina Braun for their assistance.

My family has always been, and continues to be there for me at all times. Finally, I am particularly grateful to my wife Amina El-tawil who has been incredibly supportive, understanding, and encouraging as she has been through the entire graduate experience with me.

CONTENTS

ABSTRACT	III
ACKNOWLEDGEMENT	V
CONTENTS.....	1
TABLE OF FIGURES	6
TABLES.....	8
PART ONE INTRODUCTION AND MOTIVATION.....	9
CHAPTER ONE INTRODUCTION.....	10
CHAPTER TWO MOTIVATION AND OBJECTIVES.....	15
2.1 Introduction	15
2.2 Motivation	15
2.2.1 Problem Statement.....	15
2.2.2 Towards a Mature Agent-Oriented Methodology.....	17
2.3 Research Objectives	18
2.4 Contribution of the Thesis.....	19
2.5 Chapter Summary.....	20
CHAPTER THREE LITERATURE REVIEW.....	21
3.1 Introduction	21
3.2 MAS Conceptual Framework	21
3.3 Agent Architectures	26
3.3.1 BDI Agent Architecture.....	26
3.3.2 Reactive Agent Architectures	26
3.3.3 Planning Agent Architecture	27
3.3.4 Knowledge-Based Agent Architectures	27
3.3.5 Deliberative Agent Architectures.....	28
3.4 Agent-Oriented Methodologies.....	28
3.4.1 Classification of Agent-Oriented Methodologies.....	30
3.4.1.1 Gaia Methodology.....	31
3.4.1.2 HLIM Methodology.....	33
3.4.1.3 The PASSI Methodology.....	35
3.4.1.4 MaSE Methodology	37
3.4.1.5 MAS-CommonKADS Methodology.....	38
3.5 Agent Methodologies Discussion	40
3.5.1 Advantages of Agent Methodologies	40
3.5.2 Difficulties of Agent Methodologies	41
3.6 Agent Programming Languages	41
3.6.1 Standard Programming Languages	42
3.6.2 Logic-based Languages.....	42
3.6.3 Hybrid Approaches.....	42
3.7 Agent Development Platforms and Frameworks.....	42
3.8 Chapter Summary.....	44
CHAPTER FOUR REQUIREMENTS FOR A COMPREHENSIVE AGENT- BASED SOFTWARE ENGINEERING METHODOLOGY	45

4.1 Introduction	45
4.2 Requirements for a New Methodology.....	45
4.2.1 Requirements on the level of Concepts	45
4.2.2 Requirements on the level of Models.....	46
4.2.3 Requirements on the level of Process	46
PART TWO SOLUTION	48
CHAPTER FIVE MULTI-AGENT SYSTEM DEVELOPMENT	
METHODOLOGY.....	49
5.1 Introduction	49
5.2 Assumptions.....	49
5.3 MASD Methodology.....	50
5.3.1 System Requirements Phase	51
5.3.1.1 System Scenario Model.....	51
5.3.1.2 Integrating UCMs and UCDs.....	52
5.3.1.3 Reservation Scenario Example.....	54
5.3.1.3.1 UCD of Reservation Scenario.....	54
5.3.1.3.2 UCMs of Reservation Scenario	57
5.3.2 Analysis Phase.....	61
5.3.2.1 Agent Architecture Stage	62
5.3.2.1.1 Roles Model	62
5.3.2.1.1.1 Discovering Roles	63
5.3.2.1.1.2 Determining Responsibilities of the Roles	64
5.3.2.1.1.3 Specifying Activities of Each Responsibility.....	64
5.3.2.1.2 Agent Model	65
5.3.2.1.2.1 Identifying Agents.....	66
5.3.2.1.2.2 Refining Roles	67
5.3.2.1.2.3 Agent Beliefs Model	67
5.3.2.1.2.4 Agent Goals Model.....	69
5.3.2.1.2.5 Agent Plans Model.....	71
5.3.2.1.2.5.1 Specifying Plans for each Goal	71
5.3.2.1.2.6 Agent Triggers Model.....	73
5.3.2.2 MAS Architecture Stage	75
5.3.2.2.1 Agent Interaction Model.....	75
5.3.2.2.2 Agent Relationship Model	77
5.3.2.2.3 Agent Services Model	79
5.3.3 Design Phase.....	80
5.3.3.1 Agent Container Model.....	80
5.3.3.1.1 Beliefs.....	80
5.3.3.1.2 Goals	81
5.3.3.1.3 Plans	82
5.3.3.1.4 Capabilities	83
5.3.3.1.5 Triggers	84
5.3.3.2 Inter-Agent Communication Model.....	84
5.3.3.3 Directory Facilitator Model	86
5.3.3.3.1 Directory Facilitator Mechanism	86
5.3.4 Implementation Phase	86
5.4 Chapter Summary.....	88
CHAPTER SIX CASE STUDY: CAR RENTAL SYSTEM.....	90
6.1 Introduction	90

6.2 Case Study: Car Rental System	90
6.3 System Requirement Phase	91
6.3.1 System Scenario Model.....	91
6.3.1.1 UCDs for Car Rentals System.....	91
6.3.1.2 UCMs for the Car Rental System.....	96
6.3.1.2.1 Reservation Scenario	96
6.3.1.2.2 Car Pickup Scenario.....	100
6.3.1.2.3 Car Return Scenario.....	102
6.3.1.2.4 Rental Extension Scenario	105
6.3.1.2.5 Car Service Scenario	106
6.4 Analysis Phase.....	107
6.4.1 Roles Model	108
6.4.1.1 Discovering Roles	108
6.4.1.1.2 Roles of the Car Rental System.....	108
6.4.1.2 The Agent Model	110
6.4.1.2.1 Identifying Agents for Car Rental System.....	110
6.4.1.2.2 Refining Roles.....	111
6.4.1.2.2.1 Refined Roles for Customer Agent.....	111
6.4.1.3 Beliefs Model	112
6.4.1.3.1 Beliefs of Customer Agent	112
6.4.1.3.2 Beliefs of Car Rental Clerk Agent.....	114
6.4.1.4 Goals Model.....	114
6.4.1.4.1 Identifying Agent Goals.....	114
6.4.1.4.2 Goals for Customer Agent	115
6.4.1.4.2.1 Plans for Request Reservation Goal	116
6.4.1.4.3 Goals for Car Rental Clerk Agent	118
6.4.1.4.3.1 Plans for Make Reservation Goal.....	118
6.4.1.5 Triggers model.....	120
6.4.1.5.1 Triggers Model of Customer Agent	120
6.4.2 MAS Architecture Stage	122
6.4.2.1 Agents Interaction Model.....	122
6.4.2.2 Agents Relationships Model.....	123
6.4.2.3 Agents Services Model	124
6.5 Design Phase.....	125
6.5.1 Customer Agent Container.....	125
6.5.1.1 Customer Agent Beliefs	125
6.5.1.2 Customer Agent Goals.....	127
6.5.1.3 Plans Request Reservation Goal	128
6.5.1.4 Capabilities	129
6.5.1.5 Triggers	130
6.5.2 Inter-Agent Communication Model.....	130
6.5.3 Directory Facilitator Model	133
6.6 Implementation Phase	134
6.6.1 Implementing the Case Study with Jadex	134
6.6.1.1 Starting an Agent.....	134
6.6.1.1.1 Defining Beliefs in the ADF	134
6.6.1.1.2 Defining Goals in the ADF.....	135
6.6.1.1.3 Defining Plans in the ADF.....	137
6.6.1.2 Agent Capabilities	138
6.6.1.2.1 Creating a Capability.....	139
6.6.1.3 Events	139

6.6.1.4 Agent Services Publication	140
6.7 Chapter Summary	141
PART THREE EVALUATION AND CONCLUSION	142
CHAPTER SEVEN A FRAMEWORK FOR THE EVALUATION OF AGENT ORIENTED METHODOLOGIES	143
7.1 Introduction	143
7.2 The Evaluation Framework	143
7.2.1 Models Related Criteria	144
7.2.2 Process Related Criteria.....	147
7.2.3 Supportive Feature Criteria.....	147
7.3 Evaluation Results	148
7.3.1 Models Related Criteria	148
7.3.1.1 Agent concepts.....	148
7.3.1.2 Agent Attributes.....	149
7.3.1.3 Ease of Use and Ease to Learn	150
7.3.1.4 Visualization Ability	150
7.3.1.5 Expressiveness.....	151
7.3.1.6 Consistency.....	151
7.3.1.7 Traceability and Model Derivation.....	152
7.3.1.8 Refinement.....	152
7.3.1.9 Ability to Model Agent Interactions.....	152
7.3.2 Process Related Criteria.....	153
7.3.2.1 Development lifecycle	153
7.3.2.2 Coverage of the lifecycle	153
7.3.2.3 Development Prospective	154
7.3.2.4 Domain Applicability.....	154
7.3.3 Supportive Feature Criteria.....	154
7.3.3.1 Software and Methodological Support.....	154
7.3.3.2 Open Systems Support	155
7.3.3.3 Robustness Support	155
7.3.3.4 Mobility Support	155
7.4 Discussion	155
7.5 Chapter Summary	156
CHAPTER EIGHT CONCLUSION	157
8.1 MASD Advantages.....	157
8.2 MASD Deficiencies.....	158
8.3 Discussion	158
8.4 Conclusion.....	160
8.5 Future Work.....	160
APPENDIXES.....	162
APPENDIX A: USE CASE MAPS	163
A.1 Use case Maps (UCMs).....	163
A.1.1 Where are UCMs Useful?.....	163
A.2 UCMs by Example	165
APPENDIX B: UML USE CASE DIAGRAMS	168
B.1 UML Use-Case Diagrams (UCDs).....	168
B.1.1 Use Case.....	168

B.1.2 Actor.....	169
B.1.3 Association Relationships.....	169
B.1.4 System Boundary Boxes	170
B.1.5 Useful Remarks	171
B.2 UCDs by Example.....	171
APPENDIX C: UML ACTIVITY DIAGRAMS.....	173
C.1 Activity Diagrams	173
C.2 Activity Diagrams by Example	174
APPENDIX D: FIPA-ACL.....	176
D.1 FIPA-ACL.....	176
D.1.1 FIPA Communicative Acts Library	176
D.1.2 FIPA Interaction Protocols Library	178
D.1.3 Messages in FIPA ACL.....	180
D.1.3.1 Message structure.....	180
APPENDIX E: JADEX FRAMEWORK.....	183
E.1 Jadex	183
E.2 Features	183
E.2.1 Java Based.....	183
E.2.2 FIPA Compliant	184
E.2.3 Goal-Oriented Agents	184
E.2.4 Framework.....	185
E.2.5 Development Tools	186
REFERENCES	187

TABLE OF FIGURES

Figure 3.1 Classification of Agent-Oriented Methodologies [Alonso 2004]	30
Figure 3.2 Gaia Methodology Models	32
Figure 3.3 HLIM Methodology Models	34
Figure 3.4 Models and phases of the PASSI methodology	36
Figure 3.5 MaSE models and phases [DeLoach 2004]	37
Figure 3.6 Models and Phases of the MAScommonKADS Methodology	38
Figure 5.1 MASD Methodology	50
Figure 5.2 The gap between the functional requirements and the design [Amyot 2001].	53
Figure 5.3 Use Case Diagram notations.	55
Figure 5.4 Use Case Diagrams for car rental system	55
Figure 5.5 Use Case Maps Notations.....	57
Figure 5.6 Use Case Map for Reservation Scenario.....	58
Figure 5.7 Plug-Ins for Request Reservation Stub.....	58
Figure 5.8 Plug-In for Verify Car Rentals Regulations.....	59
Figure 5.9 Plug-In for the Check Customer Demands Stub	60
Figure 5.10 Plug-Ins for Cancel Reservation Request Stub.....	61
Figure 5.11 Extracting Roles from UCMs and UML Use Cases	63
Figure 5.12 Examples of Component-Role Relations.....	64
Figure 5.13 Assigning Roles to Agents	66
Figure 5.14 Assigning Renter role to Customer agent, Rentier role to Car Rental Clerk and Manager Agents and Director Role to Car rental Manager Agent.....	66
Figure 5.15 Mapping of Agent's Roles to Goals and Plans.....	70
Figure 5.16 Notation of Interaction Diagrams.....	76
Figure 5.17 Mapping from UCMS Scenarios to Interaction Diagrams	77
Figure 5.18 Dependency Relationship Symbols	78
Figure 5.19 Dependency Diagram between Customer Agent and Reservation Agent ...	79
Figure 5.20 The Correspondence between Interaction Diagrams and FIPA Protocols .	85
Figure 6.1 Use Case Diagrams for Car Rental System.....	92
Figure 6.2 Use Case Map for Reservation Scenario.....	97
Figure 6.3 Plug-Ins for Request Reservation Stub.....	97
Figure 6.4 Plug-In for Verify Car Rentals Regulations.....	98
Figure 6.5 Plug-In for the Check Customer Demands Stub	99
Figure 6.6 Plug-Ins for Cancel Reservation Request Stub	100
Figure 6.7 Car Pickup Scenario	100
Figure 6.8 Plug-ins for Pay Rental Stub.....	101
Figure 6.9 Plug-Ins for Pay by Loyalty Points Stub.....	102
Figure 6.10 UCM for Car Return Scenario	103
Figure 6.11 Plug-Ins for the Terminate Transaction Stub	103
Figure 6.12 Plug-In for Add Loyalty Points Stub	104
Figure 6.13 UCM for Rental Extension Scenario.....	105
Figure 6.14 Plug-Ins for Request to Extend Rental Stub.....	105
Figure 6.15 Plug-ins for Manage Extension Stub.....	106
Figure 6.16 UCM Car Service Scenarios.....	107
Figure 6.17 Assigning Renter Role to Customer Agent, Rentier Role to Car Rental Clerk and Manager Agents and Director Role to Car Rental Manager Agent.	111

Figure 6.18 Interaction Diagrams between Customer Agent and Car Rental Clerk Agent	123
Figure 6.19 Dependency Diagram between Customer Agent and Car Rental Clerk Agent	124
Figure 6.20 The Correspondence Between Interaction Diagrams and FIPA Protocols	131
Figure A.1 UCM Scenario for Money Withdrawal	165
Figure A.2 UCM Scenario for Money Withdrawal with Stubs.....	166
Figure A.3 Fingerprint Plug-In for the Validate Stub.....	166
Figure A.4 Password Plug-In for the Validate Stub.....	167
Figure B.1 Use Case Diagrams Notations.....	168
Figure B.2 Includes Relationship.....	170
Figure B.3 Generalization Relationship.....	170
Figure B.4 Extends Relationship	170
Figure B.5 ATM System Use Case Diagram.....	171
Figure C.1 Notation of Activity Diagrams	174
Figure C.2 Withdraw Money from a Bank Account through an ATM Activity Diagram	175
Figure D. 1 FTPA ACL Message Structure	176
Figure D.2 FIPA Request Interaction Protocol	178
Figure D. 3 FIPA Request Interaction Protocol.....	179
Figure E.1 FIPA Agent Management	184
Figure E.2 FIPA Agent Management.....	185

TABLES

Table 5.1	Role Attributes.....	64
Table 5.2	Renter Role for Customer Component.....	65
Table 5.3	General Structure of the Agent Beliefs Model.....	68
Table 5.4	Beliefs Model for Customer Agent.....	68
Table 5.5	General Structure of the Agent Goals Model.....	70
Table 5.6	Goals Model for Customer Agent.....	71
Table 5.7	General Structure of the Agent Plans Model.....	72
Table 5.8	Reserve Car Online Plan for the Customer Agent.....	73
Table 5.9	General Structure of Agent Triggers Model.....	74
Table 5.10	Customer Agent Triggers Model.....	74
Table 5.11	Performatives for Agent Conversation Language.....	76
Table 5.12	Agent Services Model.....	79
Table 5.13	Revised agent beliefs model.....	81
Table 5.14	Revised agent goals model.....	82
Table 5.15	Reserve Car Online Plan with Plan Type Field.....	83
Table 6.1	Renter Role for Customer Component.....	109
Table 6.2	Rentier Role for Car Rental Clerk Component.....	110
Table 6.3	Refined Renter Role for Customer Agent.....	112
Table 6.4	Customer Agent Beliefs.....	113
Table 6.5	Car Rental Clerk Agent Beliefs.....	114
Table 6.6	Goals for customer agent.....	115
Table 6.7	Reserve by Phone Call Plan.....	116
Table 6.8	Reserve by E-mail Plan.....	117
Table 6.9	Reserve Car Online Plan.....	117
Table 6.10	Goals for Car Rental Agent.....	118
Table 6.11	Request Information Plan.....	119
Table 6.12	Verify Car Rental Regulations Plan.....	119
Table 6.13	Check Customer Demands Plan.....	120
Table 6.14	Triggers of Customer Agent.....	121
Table 6.15	Triggers of the Car Rental Clerk Agent.....	122
Table 6.16	Agent Services Model.....	125
Table 6.17	Beliefs of a Customer Agent.....	127
Table 6.18	Revised Customer Agent Goals.....	128
Table 6.19	Revised Reserve Car Online Plan.....	128
Table 6.20	Plan Types.....	129
Table 6.21	Common Goal Attributes (BDI flags).....	136
Table 7.1	Evaluation by Models Related Criteria.....	153
Table 7.2	Evaluation by Process Related Criteria.....	154
Table 7.3	Evaluation by Supportive Related Criteria.....	155
Table A.1	Basic Use-Case Maps (UCMs) symbols.....	164
Table D.1	Table D.1 FIPA Communicative Acts.....	177
Table D.2	FIPA ACL Message Parameters.....	181

PART ONE

INTRODUCTION AND MOTIVATION

Part one of this thesis identifies and defines the problem to be solved in this research. This part consists of four parts. The first is an introduction, which summarizes the thesis. The introduction enables the reader to gain a better understanding of the problem and provides a general view of the research work. It explains to the reader what the thesis is about and, more importantly, to justify why our research work is significant. This step is described in chapter 1.

The second part provides the motivation and the research objectives. The problem that motivated the research for this thesis is outlined and defined. An appreciation to the problem is presented and the new contribution is put into context. This step is described in chapter 2.

The third part lays out a detailed study of previous and current research. It brings the reader up to date with the latest research and development in the area. It also provides an understanding and knowledge of the present and most recent work. This step is described in chapter 3.

The last part closes by stating the requirement of the new solution, which is identified at the end of the detailed study. This step is described in chapter 4.

CHAPTER ONE

INTRODUCTION

Agent systems have created recently a great interest as a powerful new approach for designing and developing complex and distributed software systems. Agents have the ability to greatly affect the work of all of us and, consequently, it is one of the most dynamic and exciting areas in computer science today. One of the most important characteristics of these systems is the inherent ability of agents to succeed in distributed complex domains such as the Internet. Agents also have the ability to communicate with other agents using an agent communication language.

An agent is concisely defined as: *a persistent computer system capable of flexible autonomous actions in a dynamic environment.*

There are many application domains where agent technologies play an important role:

- Interoperability among information systems, where agents carry out dynamic searches for relevant information in non-local domains. Agents perform the dynamic searches on behalf of their users or on behalf of other agents. This includes retrieving, analyzing, manipulating, and integrating information available from multiple information sources.
- Electronic commerce, where agent systems support the automation of information gathering activities and sales transactions on the Internet.
- Grid computing, where agent systems enable the efficient use of resources of high-performance computing infrastructure in science, engineering, medical, and commercial applications.
- Bioinformatics and computational biology, where intelligent agents may support the coherent exploitation of the data revolution occurring in biology.

In addition, agent systems have been a source of technologies to a number of research areas, both theoretical and applied. These include distributed planning and decision-making, automated action mechanisms, communication languages, coordination mechanisms, ontologies and information agents, negotiation, and learning mechanisms. Moreover, agent technologies have drawn from, and contributed to, a diverse range of academic disciplines, in humanities, sciences, and social sciences. They also play a role in other application domains such as: monitoring and control, resource management, space, military and manufacturing application.

Multi-Agent Systems (MASs) aim to provide principles for the construction of complex distributed systems involving multiple agents and mechanisms for the coordination of independent agents' behaviors. A group of agents may work cooperatively in order to solve complex problems, which is the principle advantage of agent systems. There are many motivations for using a group of agents as a collaborative problem-solving system. They can solve problems too large for an individual agent. They can also provide the modularity of individual agents that are specialized to perform particular tasks. Multi-agent systems are concerned with the coordinated behavior of a collection of agents to achieve system-level goals.

Building multi-agent applications for such complex and distributed systems is not an easy task [Edmunson, Botterbusch, and Bigelow 1992]. Indeed, the development of industrial-strength applications requires the availability of software engineering methodologies. However, developing such complex and distributed software systems without a methodology is analogous to cooking without a recipe. Software engineers are unable to produce complex and high-quality applications in an ad-hoc fashion. Methodologies are the means provided by software engineering to facilitate the process of developing software and, as a result, to increase the quality of software products.

These methodologies typically consist of a set of methods, models, and techniques that facilitate a systematic software development process. Currently, the multi-agent system design research focuses on developing the design of full-lifecycle methodologies. Such methodologies should be able to create a multi-agent system starting with the initial specification, system requirements, and finally producing an implementation code. These methodologies should assist developers to analyze, design, and implement the agent systems.

These methodologies are different from each other in many respects such as concepts, models, software development phases, covered phases, and the supported multi-agent system properties. There are essentially three trends of classification of the agent-oriented methodologies according to the discipline on which they are based [Alonso 2004; Henderson-Sellers and Giorgini 2005]. The first is an agent-based classification, which dictates that agent-oriented methodologies are developed independently of the other traditional methodology approaches. The second classification is object-oriented based, which involves taking an existing OO methodology and extend it to support agent concepts. The last is knowledge engineering-based, which considers knowledge to be the single most important factor in organizational success of multi-agent systems.

Recently, many agent-oriented methodologies have been proposed such as: MaSE [DeLoach 2004], Prometheus [Padgham and Winikoff 2003], Tropos [Bresciani et al. 2002-2003], ODAC [Gervais 2003], Gaia [Wooldridge 2000; Zambonelli, Jennings, and Wooldridge 2003], HLIM [Elammari and Lalonde 1999], MAS-CommonKADS [Iglesias, Garrijo, Gonzalez and Velasco 1999] etc. There has been a lot of work, which involves suggestions in developing agent-oriented methodologies to cover broader software engineering lifecycle activities. These emerging methodologies attempt to exploit the key ideas behind agents at various stages of the software development lifecycle. In spite of that, the available methodologies still have the problems of traditional distributed and concurrent systems [Wood and DeLoach 2001]. This is in addition to the difficulties that arise from flexibility requirements and sophisticated interactions. Up to now, no well-established methodology exists as a development process for the construction of agent-oriented applications. Luck et al. [2003] state that there are two main technical difficulties associated with the extensive use of multi-agent systems. Firstly, there is no standard methodology that enables designers to clearly structure and construct applications as MASs. Secondly, there are no robust, broad-spectrum industrial toolkits that are flexible enough to specify and implement the characteristics of the agents involved. Furthermore, they suffer several limitations and shortcomings, which we will explain in detail in chapter 2.

We believe that the area of agent-oriented methodologies is growing rapidly and that the time has come to begin drawing together the work from various research groups with

the aim of developing the next generation of agent-oriented software engineering methodologies.

The main objective of this thesis is to develop a new agent-oriented methodology for multi-agent systems development. This methodology is considered as an organized set of guidelines that illustrate how the system agents work, cooperate, and interact with each other and with the environment that they reside within. Our approach provides a core methodology and integrates additional features into it, which are incorporated from different methodologies. In fact, we have not developing the new methodology from scratch, but we have decided to exploit existing methodologies and reuse existing technologies. We assume this methodology to be an attempt towards being comprehensive through the unification of existing methodologies by combining their strong points as well as avoiding their limitations. Our approach proposes some suggestions to form a unified methodology based on the most recognized methodologies. These suggestions, as we believe, may contribute a step towards developing the next generation of agent-oriented methodologies.

The new methodology is composed of four main phases; system requirements phase, analysis phase, design phase, and implementation phase. Chapter 5 presents a more detailed discussion of each of the four phases. The car rental system is used as a case study to describe the process of the new methodology (MASD methodology). To simplify this process, we will not describe the scenarios of the whole system. We describe only the reservation scenario as an example. The car rental system is described fully by a case study in chapter 6.

The following presents an overview of the rest of the thesis chapters categorized by the main three parts of the thesis (**Part one – Introduction and Motivation, Part two – Solution and Part three – Evaluation and Conclusion**).

Part one – Introduction and Motivation

Chapter 2 introduces the motivation and main objectives of the thesis. This chapter presents the rationale behind the development of a new multi-agent system development methodology. It presents the problem statement of this research, which defines the difficulties of the agent-oriented methodologies and provides a detailed discussion of the limitations and shortcomings of existing methodologies. It then presents the research objectives. We conclude this chapter by stating the major contributions of our research work.

Chapter 3 introduces the state of the art of the agent systems and agent-oriented methodologies. The chapter gives an overview of the concepts, rationales, hypotheses, goals and modern agents. It introduces the following:

- An overview of the rapidly evolving area of software agents from the point of view of other approaches or disciplines such as Artificial Intelligence (AI) and Software Engineering (SE).
- A historical background of agent systems.
- The terms and concepts concerned with MASs and intelligent software agents, varying from the weak notion of agency as proposed by Wooldridge [1995], to the strong notion of agency involved in defining agents as intentional systems [Wooldridge 2002].

- An overview of the agent architectures. It describes five agent architectural styles and the components they are constructed from.
- The definition of multi-agent systems. It describes two types of multi-agent systems: Closed and open multi-agent systems.
- The introduction of some of the existing agent-oriented methodologies and the classification of the methodologies according to the approach on which they are based.
- Discussion of some available agent-oriented methodologies for the development of multi-agent systems. At least one methodology of each approach is described. This chapter also points out their strengths and limitations.
- Discussion of the strengths of agent-oriented methodologies as well as the difficulties that they faced.
- An overview of the agent programming languages and agent development frameworks.

Chapter 4 presents the requirements for a novel methodology. These requirements are classified into three types of categories as follows: concepts, models (modeling techniques) and processes.

Part two – Solution

Chapter 5 introduces the new proposed methodology for multi-agent system development. The chapter starts with the assumptions (limitations) of the new methodology and then provides a detailed description for each phase of the proposed methodology as well as its models. The new methodology is called the Multi-Agent System Development (MASD) methodology. This methodology is composed of several phases such as system requirement, analysis, design, and implementation phases. This chapter uses the case study (reservation scenario of car rental system) to explain the process of MASD methodology. All phases of the methodology are demonstrated using this case study.

This chapter also describes the system requirement phase of the MASD methodology. It explains the complete detailed process of describing multi-agent systems through the case study. This phase describes the system scenario as a high-level design using well-known techniques through the system scenario model. Such techniques are called Use Case Maps (UCMs) and UML Use Case Diagrams (UCDs). The system scenario model provides high-level visual representations of the system and it is used for generating more detailed visual descriptions. In addition, it captures the behavior of a system as it appears from the point of view of an outside user.

Chapter 5 also discusses the analysis phase. The analysis phase is considered the most important process of the methodology. It explains how the agents and their roles within the system are captured. It also states how the analysis phase uses the system requirements phase (by deploying the system scenario model that is constructed by Use Case Maps and UML Use Cases) to develop agent and MAS concepts and their components. The analysis phase is composed of two main parts: agent architecture and MAS architecture. Each part is composed of a set of models. The agent architecture part describes the agent's internal structure aspects which represents the roles model, agent model, beliefs model, goals model, plans model and triggers model.

The MAS architecture part describes the MAS structure aspects, which include interactions model, agent relationships model and agent services model.

The design phase is also introduced in Chapter 5. This phase describes the process of mapping from the design to the implementation. It captures the concepts that have been developed in the analysis process and illustrates how these concepts are transferred into design specifications to be ready for implementation. This is done by identifying how to handle agent's roles, beliefs, goals and plans, as well as stating how to compose the agent capabilities into reusable agent modules. This chapter also discusses some information exchange aspects which relate to the intra- and inter-agent level and agent services.

Chapter 5 finally presents the implementation/construction phase. This phase is considered as the point in the development process when the system actually starts to construct the solution and the start of the program code writing. It creates a set of modules that have a complete set of design specifications showing how the agent system and its components should be structured and represented. The implementation process explains how the design models are handled by an agent platform called Jadex. It also presents the implementation code as Jadex proposed.

Chapter 6 introduces a complete case study for car rental systems. This case study starts by capturing the system requirements of the car rental system and ends with the implementation code. The case study has been implemented by the Jadex agent framework.

Part three – Evaluation and Conclusion

Chapter 7 introduces the evaluation framework for some well-known existing agent-oriented methodologies and compares them with the new MASD methodology. The framework provides several criteria, which were assumed after studying the common features. The study looks at common features among different methodologies used in building agents and how agent behavior is captured. The framework criteria support a number of important factors upon which the analysis and design of agents systems depend. These criteria are stated as follows: Models related criteria, process related criteria and supportive related criteria. Each criterion consists of several factors.

Chapter 8 presents the research contribution, deficiencies, discussion and conclusion of the thesis, and future work.

CHAPTER TWO

MOTIVATION AND OBJECTIVES

2.1 Introduction

This chapter discusses the motivation behind the development of a novel Multi-Agent System Development (MASD) methodology, ready to use, complete and highly expressive. This chapter discusses the motivations in the form of limitations and shortcomings, which existing methodologies face. It also presents the main research objectives. It then concludes by providing the major contributions of our research.

2.2 Motivation

Many agent-oriented methodologies and modeling languages have been proposed such as: Gaia [Wooldridge 2000; Zambonelli, Jennings, and Wooldridge 2003], MaSE [DeLoach 2004], MESSAGE [Caire 2001], Tropos [Bresciani 2003], HLIM [Elammari 1999], Prometheus [Padgham 2003], and AUML [Bauer and Odell 2005] etc. These methodologies are established to develop multi-agent systems with support tools that allow developers to create complex agent applications. They are built and specifically tailored to the characteristics of agents. However, they are still considered incomplete and suffer all the problems of traditional distributed and concurrent systems [Dastani 2004; Sabas 2002]. In addition, several limitations arise from the flexibility requirements and sophisticated interactions. Many evaluation frameworks and comparisons of the existing agent-oriented methodologies have been proposed such as [Abdelaziz, Elammari and Unland 2007; Bobkowska 2005; Sudeikat et al. 2004; Dastani et al. 2004; Silva et al. 2004; Sturm and Shehory 2003; Cernuzzi and Rossi 2002]. Most of them agree on the fact that despite the majority of the methodologies are developed based on strong foundations, they suffer from a number of limitations. These limitations are stated in detail in the following section.

2.2.1 Problem Statement

This section discusses the problem statement in the form of limitations and shortcomings that existing agent oriented methodologies suffer from. The detailed discussion of such problems is out of the scope of this thesis, consequently, no detailed discussion of such problems will be provided here. Though, a brief discussion will be provided. The discussion will not be limited to problems of a single particular methodology. Instead, it will address problems that relate to one methodology or relate to a number of methodologies. The following is a discussion of problems found during this research work:

- 1) None of the existing agent-oriented methodologies has itself established as a standard nor have they been commonly accepted [Luck et al 2003]. As long as there are no standard definitions of an agent, agent architecture, or an agent language, we could think that the existing methodologies will only be used by individual researchers to program their agent-based application using their own agent language, architectures, and theories. The lack of standard agent architectures and agent programming languages is actually the main problem to

define models and put them into operation, or providing a useful “standard” code generation. Since there is no standard agent architecture, the design of the agents needs to be customized to each agent architecture. Nevertheless, the analysis models are independent of the agent architectures. They describe what the agent-based system has to do, but not how this is done [Iglesias, Garrijo and Gonzalez 1999]. Moreover, there is no agreement on how existing methodologies identify and characterize some of the important agent aspects, such as the goals, beliefs, plans, or roles that the agents play in the system and interactions [d’Inverno 2004; Dastani 2004]. The existence of such an agreement would contribute the agent standardization.

- 2) Most of the research that examined and compared properties of agent-oriented methodologies suggested that none were completely suitable for industrial development of multi-agent systems [Tran and Low 2005; Luck et al 2003].
- 3) Most of the concepts used by the agent-oriented methodologies, like roles, responsibilities, beliefs, goals, plans, and tasks do not have formal semantics or explicit formal properties. This is an important issue when these concepts are applied, as implementation constructs need to have exact semantics [Dastani 2004].
- 4) Most of the existing methodologies suffer from a gap between the design models and the existing implementation languages [Sudeikat 2004]. It is difficult for a programmer to map the developed complex design models onto an implementation. To close this gap, a methodology should either provide refined design models that can be directly implemented in an available programming language or use a dedicated agent-oriented programming language which provides constructs to implement the high-level design concepts.
- 5) Most of the existing methodologies do not include an implementation phase. Methodologies that include an implementation phase as an essential phase of its methodology, such as the Tropos methodology, provide an explicit implementation language. This implementation language however does not explain how to implement reasoning about beliefs, goals, plans and reasoning of communication [Dastani 2004]. This leads to difficulties using the methodology. The implementation phase should describe in detail how the belief, goals, plans, and interactions are to be implemented using a specific agent programming language.
- 6) One important characteristic of agent behaviour is that the agent may play one or several roles in the system. A few of the existing methodologies support role concept. None of them takes into account that an agent may play more than one role in a system [Silva 2004]. This aspect gives the agent more flexibility and the ability to complete the work mandated. The agent can benefit from combining the goals and plans for the roles played by the agent and the latter can be exploited to carry out its work in the system.
- 7) Most of the object-oriented methodologies consider agents to be complex objects, which are not accepted by many their researchers as agents have a higher level of abstraction than objects. They also fail to properly capture the autonomous behavior of agents, interactions between agents, and organizational structures [Bush 2001]. Furthermore, complex objects cannot offer the same properties as agents do. As a result, such methodologies generally do not provide techniques to model the intelligent behaviour of agents [Jennings & Wooldridge 1999].
- 8) Most of the methodologies do not take into account the environment features. Just a few of them tried to analyze the environment, its entities, and their

interactions [Dastani 2004]. In the analysis, the methodologies studied do not consider the environmental embedding of a system. The structure of the organization in which a system will be embedded, has a large influence on the type of organizational structure of the system, at least when it interacts with more than one person.

- 9) Most existing methodologies are based on a strong agent-oriented basis. However, they do not support essential software engineering issues such as preciseness, accessibility, expressiveness, refinement, model derivation, traceability, clear definitions, and modularity. This has an adverse effect on industry acceptability and the adoption of agent technology [Dam 2003].
- 10) Confusion and ambiguity in the analysis and design phases. This is due to the absence of the holistic view of the system components, logic, cognitive and commutative aspects which should be recognized before designing the methodology models.
- 11) Some of the existing methodologies contain several misconceptions when introducing and defining the concepts and in building analysis and design concepts. This is due to the disagreement regarding agent concepts and terminology. There is in fact an extensive disagreement on the approaches that each methodology is based on. Some methodologies work on the basis of AI approaches, others work on the basis of software engineering approaches, while others use both [Sturm 2003; Dam 2003].
- 12) Many of the methodologies are incomplete. Some of the methodologies propose only the analysis and design phases, while some propose specification, analysis, and design but they do not mention anything about the tools that support the methodology. Consequently, it is too difficult to find a complete methodology [Sabas 2002].
- 13) Incomplete formality. Despite, a number of approaches for formally specifying agent system concepts have been developed such as that by Shoham [1997], Goodwin [1995], Wooldridge [1992] and Luck [1997]. Until now, there was no complete formalism for MAS concepts, which were capable to clearly describe, specify and define them in an accurate manner. They were also unable to represent the important aspects of an agent-based system such as agent beliefs, goals, actions, and interactions. This is due to the lack of agreement amongst existing methodologies as mentioned in point 11 [Luck 2004].
- 14) Open systems are not supported by the most existing methodologies. None of the methodologies allow for the dynamic addition or removal of agents, or their characteristics [Dastani 2004]. Except Gaia methodology, Gaia is extended (called Gaia hereafter) for the analysis and design of open multi-agent systems [Zambonelli, Jennings, and Wooldridge 2003].

All of the above combined provide us with a motivation to come up with a novel approach towards a comprehensive agent oriented software engineering methodology for multi-agent systems development. This novel methodology is an attempt to overcome most of the limitations stated above.

2.2.2 Towards a Mature Agent-Oriented Methodology

We will provide a new approach towards a comprehensive agent oriented software engineering methodology. The new approach is based on the exploitation of existing methodologies and reusing existing methodologies. This is done through the unification of existing methodologies by combining their strong points as well as avoiding their limitations and weaknesses. We consider such effort is similar in spirit to the one that

gave birth to Unified Modeling Language (UML). However, their approach was to build a core methodology and to integrate additional features into it from different methodologies. The integration is performed on an application-by-application basis. Our approach is different in that we endeavor to make some preliminary suggestions to form a unified methodology based on the most well known agent-oriented methodologies. These suggestions, we believe, will contribute a further step towards developing the “next generation” of agent-oriented methodologies. Such a methodology should support in sufficient depth all the following factors:

1. *Concepts*

- a) Internal properties: autonomy, mental attitudes, pro-activeness, reactivity.
- b) Social properties: methods of cooperation, teamwork, services, dependencies, agents’ relationships, communication modes, protocols, and communication language.

2. *Models*

- a) Usability criteria: clarity and understandability, adequacy and expressiveness, and ease of use.
- b) The level of effectiveness and quality of models.
- c) Technical criteria: unambiguity, consistency, traceability, refinement, and reusability.

3. *Process*

- a) Full life-cycle coverage, iterative development which allows both top-down and bottom-up design approaches
- b) Ability to represent agent behavior, agent interactions.
- c) Visualization of the system.
- d) Sufficiently detailed process steps with definitions, examples, guidelines, and heuristics.
- e) Supporting various development contexts such as reuse, prototype, and reengineering.
- f) Technical criteria: a wide range of domain applicability, support for the design of scalable and distributed applications.

2.3 Research Objectives

The objective of this thesis is to produce a new ready-to-use, highly expressive and a full-lifecycle methodology for developing multi-agent systems. This goal was supported by research examining the different ways that agents have been used in the creation of software systems. The research focused on design and analysis abstractions, looking at the system lifecycles, and creating design processes. The result of this research was the construction of a methodology for creating software systems based on multiple software agents. The limitations exhibited by the various agent-oriented methodologies led to the development of a new MASD Methodology.

The new methodology entails a comparison and evaluation framework between the existing agent methodologies and the new methodology. The framework identifies the strengths and weaknesses that will help the development and improvement of the new generation of agent-oriented methodologies. The need for such a methodology was the most significant motivation driving the development of the proposed methodology.

2.4 Contribution of the Thesis

The main contribution of this thesis is to build a new methodology for the development of multi-agent systems. This methodology is expected to be a solid and reliable guide in building and developing such systems.

The MASD methodology is based upon other research including the previous existing methodologies. It aims to develop a complete life-cycle methodology for designing and developing MASs. In addition, MASD is considered as an attempt to solve some of the problems that was mentioned precisely. We state the solution as follows:

The MASD methodology was established based on three fundamental aspects: concepts, models, and process. These three aspects are considered as a foundation for building a solid methodology. The concepts are all the necessary MAS concepts that should be available in order to build the models of the new methodology in a correct manner. The models include modeling techniques, modeling languages, a diagramming notation, and tools that can be used to analysis and design the agent system. The process is a set of steps or phases describe how the new methodology works in detail.

In addition, the MASD methodology bridges the gap between the design models and the existing implementation languages. It provides refined design models that can be directly implemented in an available programming language or use a dedicated agent-oriented programming language which provides constructs to implement the high-level design concepts such as Jadex, JADE, JACK, etc. In addition, it helps developers to map the developed complex design models into implementation constructs.

Furthermore, the MASD methodology proposes an important concept called triggers and relies heavily on agent roles. The role concept is considered one of the most important aspects that represent the agent behaviour. Therefore, MASD assumes each agent can play one or more roles in the system. The trigger concept is also considered as an important aspect that represents the agent reactivity. Furthermore, MASD considers the social agent aspects. This is by utilizing well-known techniques such as use case maps, which enable developers to identify social aspects from the problem specification. Therefore, MASD assumes the agent society architecture should be derived from the problem specification that will lead to the best-suited architecture.

Moreover, MASD methodology is developed based on the essential software engineering issues such as preciseness, accessibility, expressiveness, domain applicability, modularity, refinement, model derivation, traceability, and clear definitions.

In addition, the MASD methodology provides a plain and understandable development process through the methodology phases. It captures the holistic view of the system components, and commutative aspects, which should be recognized before designing the methodology models. This is by using well-known techniques such as UCMs and UML UCDs.

2.5 Chapter Summary

In this chapter, we discussed the motivation behind the development of a novel multi-agent system development methodology and the various limitations illustrated by the various existing methodologies in the area. The chapter concluded with the research objectives and the major contributions of our research work in this area.

CHAPTER THREE

LITERATURE REVIEW

3.1 Introduction

The aim of chapter 3 is to provide an overview of the rapidly evolving area of multi-agent systems (MASs) and its related methodologies. This leads to a discussion of what makes an agent-oriented methodology that can be used to build a MAS. The chapter starts with discussing the common terms and concepts that is used in the thesis. In this chapter, literature concerning agent systems and agent-oriented methodologies is reviewed in detail. The aim is to establish the characteristics of agent-oriented methodologies, and see how these characteristics are suited to develop multi-agent systems. The current research of agent-oriented methodologies is examined giving a clearer picture of their application domain, advantages and limitations.

3.2 MAS Conceptual Framework

Before introducing the chapter 3, we have to define the common terms and concepts that are used in this thesis, enabling the reader to understand the next parts of the thesis. The concepts underlying multi-agent systems and the associated agent terminology are not universally agreed upon [d’Inverno and Luck 2004; Dastani et al. 2004]. However, there is adequate agreement to make it valuable for us to summarize the commonly agreed upon terms in this chapter in order to clarify them for the later chapters. This conceptual framework introduces the MAS concepts that the new methodology relies on.

The topic “software agents” has become one of the most striking topics in computer science research. The term “Software agent” leads to a wide argument of what a software agent is, and of how it could be clearly distinguished from a program.

Examining the question, “What is a software agent?” raises many arguments about what a software agent is, and what the difference between a software agent and computer program is. Researchers have proposed many definitions for the concept of a software agent. Each of them introduced his/her definition according to their point of view. Some of them concentrated on artificial intelligence approaches, others concentrated on software engineering approaches. We concentrate on the definitions that are well known and most accepted by agent researchers such as M. Wooldridge and N. Jennings etc.

A general definition of a software agent is that it is a computer program that exhibits the characteristics of an agency or a software agency. According to Krupansky's Foundations of Software Agent Technology the software agent is defined as: “*A software agent (or autonomous agent or intelligent agent) is a computer program which works toward goals (as opposed to discrete tasks) in a dynamic environment (where change is the norm) on behalf of another entity (human or computational), possibly over an extended period of time, without*

continuous direct supervision or control, and exhibits a significant degree of flexibility and even creativity in how it seeks to transform goals into action tasks.” [Krupansky 2008]

Here we present another software agent definition, which clearly distinguishes a software agent from any other program. Wooldridge and Jennings [1995] proposed two notions of agency; a weak notion and a strong one. A weak notion of agency “*is that of hardware or more frequently software-based computer system that provides the following properties:*

1. *Autonomy: is when agents operate without the direct intervention of humans or others, and have some kind of control over their actions and internal state;*
2. *Social ability: is the ability of agents to interact with other agents (and possibly humans) via some kind of agent-communication language;*
3. *Reactivity: The ability of agents to perceive their environment and respond in a timely fashion to changes that occur in it. Here, the agent environment may be the physical world, a user via a graphical user interface, a collection of other agents, the Internet, or perhaps all of these combined.*
4. *Pro-activeness: is when agents do not simply act in response to their environment, but they are also able to exhibit a goal-directed behaviour by taking the initiative.”*

A strong notion of agency is also widespread in artificial intelligence. In addition to the weak notion, the strong notion also uses mental components such as belief, desire, intention, and knowledge and so on. This definition illustrates autonomy of an agent, sensing and acting on a finite environment that an agent is a part of [Wooldridge 2002].

Many agent definitions are proposed by software agents research Maes [1995], Russel [1995], Riecken [1994], and Wooldridge and Jennings [1995]. Most of these definitions were based on certain situations, certain conceptions, or to solve certain problems according to a researcher’s point of view. As an attempt to cover most of the existing agent definition patterns, we conclude the following agent definition and its related concepts to be the foundation for the new methodology. These definitions are stated as follows:

Agent: A persistent computer system that carries out some set of tasks on behalf of a user or a computer system and is capable of:

1. Functioning with some degree of autonomy (autonomy means the agent ability to work with minimum intervention by the real user. The autonomous agents have control over their tasks and resources and will take part in cooperative activities only if they chose to do so).
2. Interacting with others (humans or agents) via specific agent communication language.
3. Perceiving its environment through sensors, acting on the environment, and reacting to the changes of the environment through effectors.
4. Employing its knowledge to make decisions.
5. Cooperating with others (humans and agents) either by negotiation or coordination to achieve common goals.
6. Realizing its goals by performing suitable roles and following suitable plans.
7. Gaining knowledge from experience to store the successful plans.
8. Being adaptable with the environment changes by responding in a timely fashion.
9. Having initiative (self starting).

Agent role: A set of actions and activities that are assigned to, or expected of an agent to be able to perform in the system. In other words, a role represents an agent behavior that is recognized, providing a means of identifying and placing an agent in a system. The distinction between an agent and a role is that an agent model describes characteristics that are inherent to an agent, whereas a role describes characteristics that an agent takes on. For each agent there is at least one role that should be performed in the system. For each role, there is at least one responsibility that should be performed by this role. For each responsibility, there exists a trigger, which is possibly triggering an action that belongs to the agent capabilities. **Responsibilities** of a role represent the main activities or tasks that the role performs in order to realize the objectives in the system.

Agent knowledge: What each agent knows about the environment state, but also what each agent knows about other agents. Agent knowledge represents the informational state of the agent about the environment including itself and other agents. It includes agent **beliefs** and **goals**.

Agent beliefs: Facts that are believed to be true about the working environment. An agent's beliefs are knowledge, which constitutes a description of the world. An agent's beliefs may be taken to explicitly represent the agent's working environment or even about the agent itself or other agents. Using the term belief rather than knowledge recognizes that what an agent believes may not necessarily be true and in fact may even change in the future.

Agent goals: Goal is defined as an end state, something to be achieved. It describes, "*What is to be done*". It is the destination itself and not a recipe for how to reach that destination. Agent goals are informational states of what it is planned to be achieved. The goals represent a mechanism, which leads the agent to achieve its tasks in an orderly and smooth way. In order to describe the goal the following two questions need to be answered. When are goals initiated or started? When are goals considered satisfied? The goal is started or initiated when its precondition(s) is satisfied. The goal is considered satisfied if and only if at least one of its plans is satisfied then its postcondition(s) is satisfied. These pre- and postconditions are considered the beliefs of the agent. The agent goals are classified into two types of goals **long-term** and **short-term** goals. **Long-term** goals are ones that the agent will achieve over a longer period. Long-term goals often are the most meaningful and important goals. One problem, however, is that the achievement of these goals is usually far in the future. Therefore, the agents should stay focused and maintain a positive attitude towards reaching these goals. **Short-term** goals are ones that the agent will achieve in the near future. Long-term goals can be decomposed to hierarchical sub-short-term goals. Short-term goals will move the agent along towards its long-term goals. Identifying the following short-term goals will help the agent to create a clear picture of where it is going.

Plans: An agent's view of the way a modeled agent will achieve its goals. A plan is an organized set of tasks the agent will do to achieve its goals. Each plan is composed of a set of tasks. These tasks will implement the plan and complete the required work.

Task: "*An atomic piece of work to be done*". It identifies how things are to be done and is clearly seen as an atomic work unit. A Task represents the miniature action that is performed by the agent, which cannot be decomposed into sub-actions.

Agent decision: An agent is capable to decide what actions to perform based on its plans, knowledge, and beliefs. An agent decides to perform actions that will change the environment situation, and by doing so, the agent's goals are committed to be satisfied.

Reactive agent: An agent that reacts to incoming events perceived in the environment. A reactive agent answers to an event by a pre-defined action.

Proactive agent: agents do not simply act in response to their environment; they are able to exhibit goal-directed behaviour by taking the initiative. An agent generates goals, tries to achieve them, and does not depend on events occurring in the environment.

Environment: A set of components that describe all the features of the system and its behavior. These components affect each other. These components are stated as follows: the agents that act on this environment, the events that happen in the environment, the interactions that could take place between the agents, and the dependencies between agents in the system. In fact, the environment constitutes the MASs.

Events: Actions that happens at a given place and time. The action uniquely identifies the event. Location is the place where the event happens. Time is that time when the event happens. They are all perceived and afterwards processed by agents and may launch or trigger plans or goals that should be selected to achieve. An agent may react to events that change its knowledge. Events may change the agent's knowledge because its perception of the environment has changed. A triggering event defines which events may lead to the execution of a particular plan in order to achieve a particular goal.

Triggers: Represent incoming information from the environment to the agent. The agent reacts according to this information in terms of actions. An agent perceives its environment through sensors that describe triggering information. This triggering information could be events or agent belief changes about the state of the environment. These events or agent belief changes trigger the agent to do actions that may update the agent's knowledge, known as beliefs and goals.

Agent interactions: The way in which agents exchange information. This exchange amounts to a message passing from one agent to many agents or humans. Interaction enables agents to negotiate and coordinate in achieving their tasks or common goals. These interactions are managed by communication acts (messages) and organized by communication protocols.

Message: A unit of information or data that is transmitted from one agent to another. A message can be defined as any information sent as an agent, which interacts with another.

Protocol: A sequence of rules, which guide the interaction that take place between several agents. These rules determine the format and transmission of messages exchanged between agents. These rules define what messages are possible for any particular interaction state. The set of possible messages is finite.

Agent services: A service is a task that an agent is potentially willing to perform on behalf of other agents. A set of services is associated with each agent. For each service that may be performed by an agent, it is necessary to specify its properties such as name, cost, etc. An agent possesses skills (services), which the agent can offer to other agents.

Agents' relationships: Denotes the degree of influence agents have over each other. They allow us to construct management hierarchies (i.e. who is the boss of who). For example, an **agent dependency relationship** is a relationship between two agents, a dependee, and a dependant. The dependant agent depends on another agent (the dependee) to do or provide something (dependum) in order that the dependant may achieve some goal.

Multi-Agent System (MAS): A system composed of several agents, capable of reaching goals that are difficult to achieve by an individual agent system. A multi-agent system is a system showing the following characteristics [Jennings 1998]:

- Each agent has incomplete capabilities to solve a problem.
- There is no global system control.
- Data is decentralized.
- Computation is asynchronous.

When several agents interact, they may form a multi-agent system. Characteristically such agents will not have all data or all methods available to achieve an objective and thus will have to collaborate with other agents. In addition, there may be little or no global control and thus such systems are sometimes referred to as “swarm systems”. As with distributed agents, data is decentralized and execution is asynchronous. MASs evolved from Distributed Artificial Intelligence (DAI), Distributed Problem Solving (DPS), and Parallel Artificial Intelligence (PAI), thus inheriting all characteristics from DAI and Artificial Intelligence (AI). Generally, multi-agent systems can show plainly self-organization and complex behaviors.

There are two types of agent systems: Closed multi-agent systems and open multi-agent systems.

Closed multi-agent systems: based on static design with components and functions, which are required to be known in advance. In such systems, there is a common language for communication between agents. Each agent is developed as an expert in a particular area has the ability to solve problems, skills, and knowledge. For example, MAS is organization built to contain a group of agents who represent different departments within the organization. Each of these agents has different skills and roles.

Open multi-agent systems: Often do not have static design beforehand there are only independent agents inside the system. Agents would not necessarily know the experience of other agents or services they offer. Therefore, it is requested that there should be a mechanism to identify agents. Agents may be uncooperative, malicious and unreliable in open systems. An example of open systems is the e-commerce market where it is not necessary for agents representing clients to look for providers in order to obtain services or products they need. This is often done through mediator

agents and brokers specially designed for this purpose and who are working as a directory.

3.3 Agent Architectures

Maes proposes agent architecture as “a particular methodology for building agents. It specifies how the agent can be decomposed into the construction of a set of component modules and how these modules should be made to interact” [Maes 1991]. Agent architectures present a higher level of abstraction for building and viewing agent systems. A numeral of existing agent architectures were reviewed, decomposed using object-oriented techniques, and then classified based on their components, connectors, and overall structural pattern. Five agent architectural styles were found using this approach: Belief Desire Intention (BDI), reactive, planning, knowledge-based, and deliberative. The following sections review the most common types of agent architectures and the components they are constructed from.

3.3.1 BDI Agent Architecture

The BDI architecture is one of the most well-known and studied software agents’ architectures [Georgeff, Pell, Pollack, Tambe, and Wooldridge 1998]. This architecture consists of four basic components: beliefs, desires, intentions, and plans. In this architecture, the agent’s beliefs represent information that the agent has about the world, which in many cases may be incomplete or incorrect [d’Inverno, Kinny, Luck and Wooldridge 1997]. The content of these beliefs can be anything from knowledge about the agent’s environment to general facts an agent must know in order to act rationally. The desires of an agent are a set of long-term goals, where a goal is typically a description of a desired state of the environment. An agent’s goals simply represent some desired end state. These goals may be defined by a user or may be adopted by the agent. New goals may be adopted by an agent due to an internal state change in the agent, an external change of the environment, or because of a request from another agent. State changes may cause goals or plans to be triggered or new information to be inferred that may cause the generation of a new goal. Requests for information or services from other agents may cause an agent to adopt a goal that it currently does not possess. An agent’s desires provide it with motivations to act. When an agent chooses to act on a specific desire that desire becomes an intention of the agent. The agent will then try to achieve these intentions until it believes the intention is satisfied or the intention is no longer achievable [d’Inverno, Kinny, Luck, and Wooldridge 1997]. The intentions of an agent provide a commitment to perform a plan. Although not mentioned in the acronym, plans play a significant role in this architecture. A plan is a representation outlining a course of action that, when executed, allows an agent to achieve a goal or desire.

3.3.2 Reactive Agent Architectures

Perhaps the simplest among the most widely used agent architectures are reactive architectures. Wooldridge and Jennings [1995] describe a reactive architecture as an architecture that does not have a central world model and does not use complex reasoning. Unlike knowledge-based agents that have an internal symbolic model from which to work, reactive agents act by stimulus-response to environmental states. The

agent perceives an environmental change and reacts accordingly. Reactive agents can also react to messages from other agents.

Although reactive agents are basic and can only perform simplistic tasks, they do form a building block from which other, more complex agents can be built. By adding a knowledge base to a simple reactive agent, the agent becomes capable of making decisions that take into account previously encountered state information. By adding goals and a planning mechanism, we can create a rather complex goal directed agent. Although complex patterns of behavior can be developed using reactive agents, their primary goals usually consist of being robust and having a fast response time. Most agent architectures contain a reactive component of some kind. However, they are not actually truly reactive agents. Majority of reactive architectures can be modeled using a basic “IF-THEN” rule structure.

3.3.3 Planning Agent Architecture

A number of researchers present different definitions for planning, but all result in the same essential facts. Planning is the process of formulating a list of actions in order to achieve a specified goal [Pollack 1992]. In artificial intelligence, a planner uses knowledge about the actions it may perform and their consequences. It uses this as well as knowledge about the environment, to formulate a list of acceptable state transforming operators that can transform the agent from an initial state into a goal state. As seen in BDI, planning architectures are usually embedded in other agent architectures to determine the actions that an agent will perform. Within a given agent architecture, plans may be either synthesized dynamically or predefined in advance and placed in a plan library. In general, plans come in two types; total order and partial order. Total order plans simply consist of a list of steps that an agent must follow to accomplish a set goal. These steps have a definite order that must be followed for the goal to be achieved. Partial ordered plans may have some steps ordered while the order of other steps is arbitrary and inconsequential to reaching the goal. At one more level of abstraction, plans can be fully or partially instantiated. The steps of a plan are generally operators containing parameters that need to be defined to a set value in order for the operator to function. A fully instantiated plan is one in which all of these parameters are defined to a set value. Russell and Norvig [1995] state that a plan is a formally defined data structure that contains the following components:

- A set of plan steps. Each step is one of the operators of the problem.
- A set of step ordering constraints.
- A set of variable binding constraints.
- A set of causal links to record the purpose(s) of steps in the plan

3.3.4 Knowledge-Based Agent Architectures

Even though the BDI architecture has a knowledge base, a large number of architectures that exist are built around a centralized knowledge store. In general, these are referred to as knowledge-based or expert systems. Knowledge-based systems use data structures consisting of explicitly represented problem-solving information. This knowledge can be viewed as a set of facts about the world. Three aspects of knowledge-based systems, which make them powerful, are:

1. They can accept new tasks in the form of explicitly described goals.

2. They can achieve competence quickly by being told or learning new knowledge about the environment.
3. They can adapt to changes in the environment by updating the relevant knowledge. [Russell and Norvig 1995]

In general, knowledge-based systems represent knowledge using a formal declarative language. The use of declarative language allows knowledge to be added or deleted from the knowledge base quickly and easily without affecting the rest of the system. Using a declarative language such as first-order logic also allows new information to be derived from the current knowledge stored in the system using inference mechanisms. An inference mechanism can perform two actions. First, given a knowledge base, it can generate new sentences that are necessarily true, given that the old sentences are true. Second, given a knowledge base and a sentence, it can determine whether the sentence was generated by the knowledge base or not [Russell and Norvig 1995]. The relation just described between sentences is called entailment and is used a great deal in knowledge-based systems.

3.3.5 Deliberative Agent Architectures

The deliberative agent architecture contains an explicitly represented, symbolic model of the world. Decisions (for example about what actions are to be performed) are made via logical reasoning, based on pattern matching and symbolic manipulation [Genesereth and Nilsson 1987]. In order to build an agent in this way, there are at least two important problems that need to be solved:

1. The transduction problem: that of translating the real world into an accurate, adequate symbolic description, in time for that description to be useful.
2. The representation/reasoning problem: that of how to symbolically represent information about complex real-world entities and processes, and how to get agents to reason with this information in time for the results to be useful.

3.4 Agent-Oriented Methodologies

In order to be able to perform a comprehensive literature review for the agent-oriented methodologies, “what the meaning of the methodology is” needs to be precisely defined before starting this discussion. A good methodology should provide the models for defining the elements of the multi-agent environment (agents, objects and interactions). A good methodology should also provide the design guidelines for identifying these elements, their components and the relationships between them. Any good methodology aims to provide a set of guidelines that covers the whole lifecycle of the system development. The guidelines should cover both the technical as well as the management aspects.

Agent systems have been increasingly recognized as the next important software engineering approach. Methodologies are the means provided by software engineering to facilitate the process of developing software and, as a result, to increase the quality of software products. By definition, a software engineering methodology is:

“A collection of procedures, techniques, tools and documentation aids which will help the systems developers in their efforts to implement a new information system. A methodology will consist of phases, themselves consisting of sub-phases, which will guide the systems developers in their choice of

techniques that might be appropriate at each stage of the project and also help them plan, manage, control and evaluate information system projects” [Avison and Fitzgerald 2003].

It is also important for a methodology to provide notations and modeling techniques, which allow the developers to model the target system and its environment. **Notations** are a technical system of symbols used to represent elements within a system. A **modeling technique** is a set of models that depict a system at different levels of abstraction and the different aspects of the system. Furthermore, an agent methodology should support software engineering issues such as: preciseness, accessibility, expressiveness, modularity, domain applicability, and scalability. **Preciseness** makes sure that the semantics of modeling techniques of the methodology are unambiguous in order to avoid misinterpretation of the developed models by those who use it. **Accessibility** is the understandability of the modeling techniques for both experts and novices. **Expressiveness** is the ability of the methodology to express the system as whole. It represents the following aspects of the system: structure; encapsulated knowledge; ontology; data flow; control flow; concurrent activities; resource constraints (e.g., time, CPU and memory); the physical architecture; agents’ mobility; interaction with external systems; and the user interface definitions. **Modularity** is the ability to express the methodology in stages. That is, when new specification requirements are added, there is no need to modify previous parts, and these may be used as a part of the new specification. **Domain Applicability** is the suitability of the methodology for a particular application domain (e.g. real-time, information systems). **Scalability** is the ability of the methodology or subsets thereof, to be used to handle various application sizes.

In addition to the methodology, there are also tools that support the use of such methodologies. For example, diagramming editors help developers draw symbols and models, which are described in the methodology. The Rational Unified Process (RUP) is a good example of a software engineering methodology [Kruchten 2000]. It uses the notation described in the Unified Modeling Language (UML) [Booch 1998] and its typical tool support is called “Rational Rose”.

A robust methodology needs to contain sufficient abstractions to entirely model and support agents and MASs. Therefore, software engineers should use agent-oriented concepts to describe the methodology. In turn, this can be used to build agent-oriented systems and MASs. Arguably, simple extensions of object-oriented methodologies to represent agent concepts are highly restricted by object concepts. Thus, an agent-oriented methodology needs to concentrate on an organized society of agents playing roles within an environment. This society of agents is interacting according to protocols determined by agents within the system.

There are a large number of agent-oriented methodologies available [Henderson-Sellers 2005]. Several efforts were directed to studying most of these existing methodologies at a more detailed level [Arazy and Woo 2002; Castro et al. 2003; Dam and Winikoff 2002-2003; Juan, Sterling and Winikoff 2002; Sabas, Delisle and Badri 2002; Sturm and Shehory 2003 etc.]. A detailed discussion of existing methodologies is given in section 3.4.1 where related research is examined.

3.4.1 Classification of Agent-Oriented Methodologies

Agent-oriented methodologies have several roots. They are classified according to the approach or discipline upon which they are based. A common property of these methodologies is that they are developed based on the approach of extending existing methodologies to include the relevant aspects of agents. They are broadly classified into three categories: agent-based methodologies, object oriented-based methodologies and their extensions, and knowledge engineering-based methodologies [Alonso 2004; Henderson-Sellers and Giorgini 2005]. Figure 3.1 illustrates the classifications of agent-oriented methodologies.

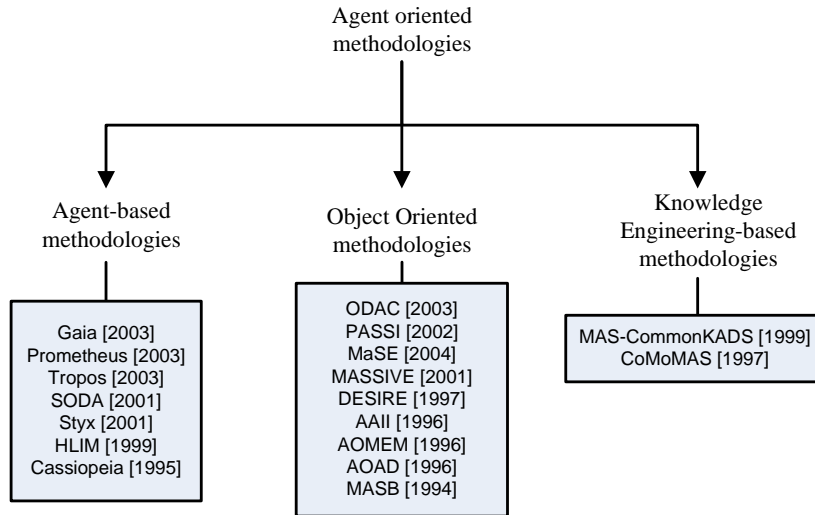


Figure 3.1 Classification of Agent-Oriented Methodologies [Alonso 2004]

Agent-based methodologies: There are several methodologies that belong to this category such as: Gaia [Wooldridge 2000; Zambonelli, Jennings, and Wooldridge 2003], HLIM [Elammari and Lalonde 1999], Tropos [Brescian 2003], Prometheus [Padgham and Winikoff 2003], SODA [Omicini 2001], Styx [Bush 2001], and Cassiopeia [Collinot 1995]. The developers of such methodologies urge that the agent concept should be established without dependency on other traditional methodologies, such as object-oriented methodologies. The main reason is the inherent differences between the two entities; agents, and objects. This is because agents have a higher level of abstraction than objects. Object-oriented approaches cannot offer the same properties as agents do. They also fail to properly capture the autonomous behavior of agents, interactions between agents, and organizational structures [Bush 2001]. In fact, the notions of autonomy, flexibility, and proactiveness can hardly be found in traditional object-oriented approaches [Odell 2002]. As a result, object-oriented methodologies generally do not provide techniques to model the intelligent behaviour of agents [Jennings & Wooldridge 1999]. Therefore, there need to be software engineering methodologies, which are specially tailored to the development of agent-based systems.

Object oriented-based methodologies (Extensions of object-oriented methodologies): The agent-oriented methodologies which belong to this category either extend existing object-oriented methodologies or adapt them to the aim of agent-oriented software engineering. The examples of such methodologies are: ODAC [Gervais 2003], MaSE [DeLoach 2004], MASSIVE [Lind 2001], DESIRE [Brazier

1997], AAIL [Kinny, Georgeff, and Rao 1996], AOMEM [Kendall 1996], AOAD [Burmeister 1996] and MASB [Moulin 1994]. Some researchers present several reasons for following this approach. Firstly, the agent-oriented methodologies, which extend the object-oriented approach, can benefit from the similarities between agents and objects. Secondly, they can capitalize on the popularity and maturity of object-oriented methodologies. In fact, there is a high chance that they can be learnt and accepted more easily. Finally, several techniques such as use cases and class responsibilities card (CRC), which are used for object identification can be used for agents with a similar purpose (i.e. agent identification) [Iglesias, Garrijo and Gonzalez 1999].

Knowledge Engineering-based methodologies (Extensions of Knowledge Engineering (KE) techniques): There are, however, some aspects of agents that are not addressed in object-oriented methodologies. For instance, object-oriented methodologies do not define techniques for modeling the mental states of agents. In addition, the social relationship between agents can hardly be captured using object-oriented methodologies. These are the arguments for adapting KE methodologies for agent-oriented software engineering. They are suitable for modeling agent knowledge because the process of capturing knowledge is addressed by many KE methodologies [Iglesias, Garrijo, and Gonzalez 1999]. Additionally, existing techniques and models in KE such as ontology libraries, and problem solving method libraries can be reused in agent-oriented methodologies. Examples of such methodologies are: MAS-CommonKADS [Iglesias, Garrijo, Gonzalez and Velasco 1999] and CoMoMAS [Glaser 1997]

Agent-oriented methodologies should assist the developer in making decisions about the aspects of the analysis, design, and implementation of the agent systems. Some methodologies focus on inter-agent aspects, while others focus on intra-agent aspects. Finally, some methodologies explicitly deal with the environment while others do not. These methodologies differ from each other in many respects. They differ on the software development phases they capture in analysis, design, and implementation phases. In addition, they differ in their premises, covered phases, models, concepts, and the supported multi-agent system properties.

Among these starting points, five dedicated agent-oriented software methodologies were chosen and further reviewed. The selected methodologies are Gaia [Wooldridge 2000; Zambonelli, Jennings, and Wooldridge 2003], HLIM [Elammari and Lalonde 1999], PASSI [Burrafato 2002], MaSE [DeLoach 2004], and MAS-CommonKADS [Iglesias, Garrijo, Gonzalez and Velasco 1999]. These methodologies were chosen because they are based on different approaches. This selection was done in order to choose from all disciplines. From within the same approach the more commonly known methodologies were selected. The existing approaches are outlined in figure 3.1.

3.4.1.1 Gaia Methodology

The Gaia methodology [Wooldridge 2000; Zambonelli, Jennings, and Wooldridge 2003] was developed by Wooldridge et al. for the analysis and design of agent systems and was extended to support open multi-agent system in 2003 by Zambonelli et al. Gaia is a general methodology that supports both levels of micro and macro development of agent systems. The micro level relates to the agent structure while the

macro level relates to the agent society and organizational structure. Gaia includes an analysis and design phase but does not explicitly support an implementation phase.

Gaia starts with the analysis phase as is given in figure 3.2. It aims to collect and organize the specification which is the basis for the design of the computational organization. It then continues with the design phase, which aims to define the system's organizational structure. The definition is in terms of the system's topology and control system in order to identify the agent model and the service model. Gaia consists of two main phases: the analysis phase and design phase.

The analysis phase is the set of requirements that are identified. It aims to understanding the system and its structure. It includes: the environmental model, preliminary role model, preliminary interaction model, and organizational rules model.

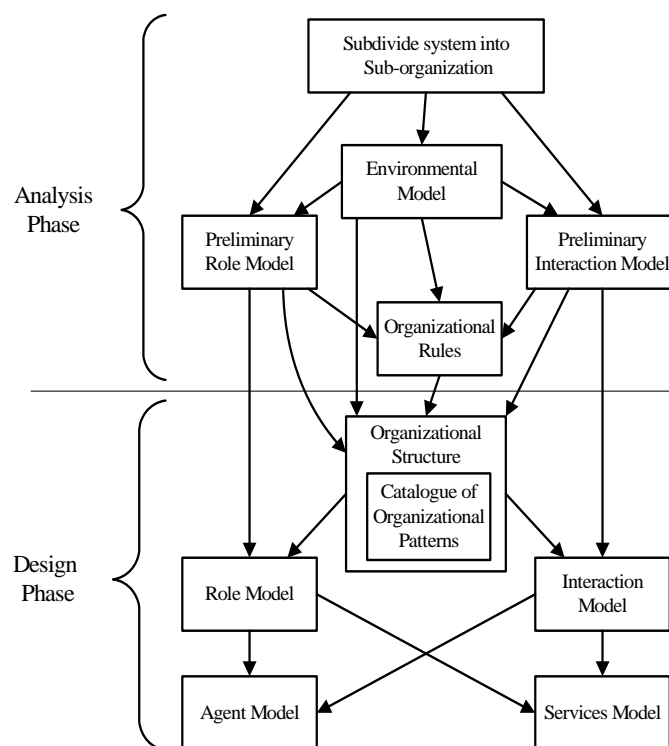


Figure 3.2 Gaia Methodology Models

The environmental model aims to make the characteristics of the environment explicit in which the multi-agent system will be engaged.

The preliminary role model specifies the key roles in the system and describes them in terms of permissions and responsibilities.

The preliminary interaction model captures the dependencies and relations between roles by means of protocol definitions. Gaia is only concerned with the society level; it does not capture the internal aspects of agent design.

The organizational rules model captures the basic functionalities required by the organization, as well as the basic interactions and roles.

The design phase includes: organizational structure, agent model, role model, interaction model, and service model.

The organizational structure captures the catalogue's organizational patterns and involves considering: (i) the organizational efficiency, (ii) the real-world organization (if any) in which the MAS is situated, and (iii) the need to enforce the organizational rules.

The role and interaction models are the completion of the preliminary role, and interaction model. This is based upon the adopted organizational structure and involves separating, whenever possible, the organizational-independent aspects (detected from the analysis phase) from the organizational-dependent ones (derived from the adoption of a specific organizational structure). This separation promotes a design-for-change perspective by separating the structure of the system (derived from a contingent choice) from its goals (derived from a general characterization).

The agent model is concerned with identifying the agent classes that will make up the system and the agent types that will be instantiated from these classes.

The service model is concerned with identifying the services associated with a role. It identifies the main services intended as coherent blocks of activity in which agents will engage. These services are required to realize the agent's roles, and their properties.

Despite that the Gaia methodology was developed based on strong software engineering approaches [Henderson-Sellers 2005], the Gaia developer's state some limitations as follows:

- The methodology does not directly deal with particular modeling techniques. It proposes, but does not commit to, specific techniques for modeling (e.g., roles, environment, and interactions).
- It does not directly deal with implementation issues.
- It does not explicitly deal with the activities of requirements capturing and modeling, and especially with early requirements engineering.

3.4.1.2 HLIM Methodology

The HLIM methodology was developed in 1999 by M. Elammari and W. Lalonde and allows the development of agent-based systems to form user requirements. The methodology models the external and internal behavior of agents. It also provides a means for both the visualization of the behavior of systems' agents and the definition of how the behavior is achieved. The methodology provides a systematic approach for generating system definitions from high-level designs, which can be implemented. The methodology captures effectively the complexity of agent systems, agents' internal structure, relationships, conversations, and commitments.

Figure 3.3 shows the model of the HLIM methodology. The HILM methodology consists of two phases: the discovery phase and the definition phase.

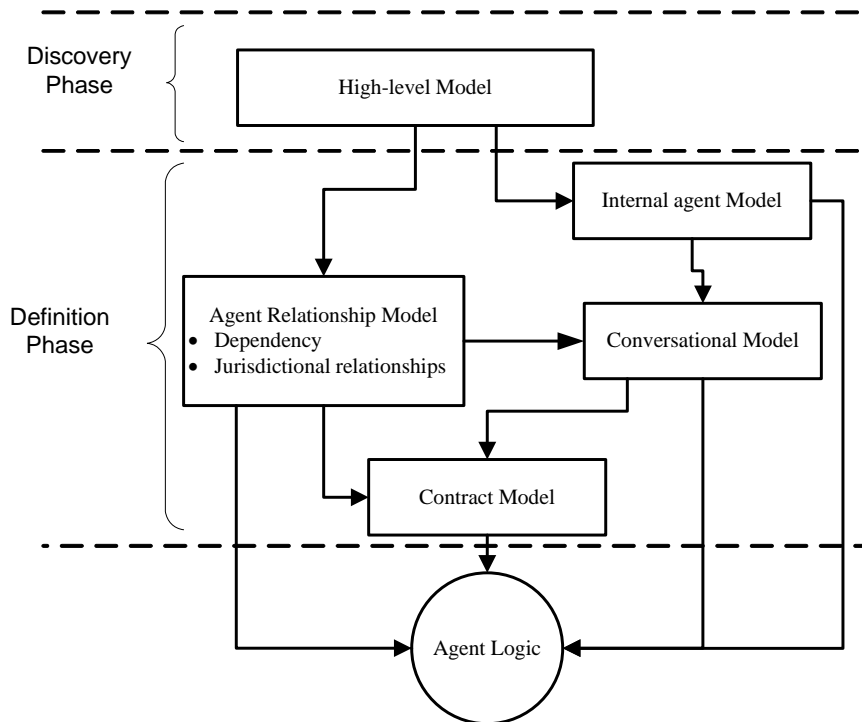


Figure 3.3 HLIM Methodology Models

The discovery phase includes the high-level model. The discovery phase is an exploratory phase that leads to the high-level model definition. The agents are discovered and their high-level behaviour is identified.

The high-level model identifies agents and their high-level behaviour. It gives a high-level view of the system and is considered as a point of commencement for providing the details of other models. It is constructed by tracing application scenarios that describe functional behaviour, discovering agents and behaviour patterns along the way. This model includes three major sources of information: Documentation defining operational aspects of the model such as preconditions and postconditions of scenarios, the responsibilities of agents, and the behaviour of the system at the level of collaborating agents achieving a specific system purpose.

The definition phase produces intermediate models that facilitate the implementation of agent systems. The high-level model, supplemented by other information, is used to generate these models. These models express the full functional behaviour of an agent system by identifying aspects of agents such as goals, beliefs, plans, jurisdictional and dependency relationships, contracts, and conversations. It includes the internal agent model, the relationship model, the conversational model, and the contract model.

The internal agent model describes the agents in the system in terms of their internal structure and behaviour. It is derived directly from the high-level model. This model describes the internal structure of the agents discovered in the high-level model. It also captures agent aspects such as goals, plans and beliefs.

The agent relationship model describes inter-agent dependencies and jurisdictional relationships. The relationship diagrams are derived from the

coordination expressed in the high-level model and from responsibilities and preconditions. Coordination is captured in the high-level model by path segments that connect two agents. Analysis of responsibilities and preconditions may lead to the discovery of dependencies. The inter-agent dependencies are represented by the dependency diagram, which explains how an agent provides a service to another agent that requires that service. The jurisdictional relationships are represented by the jurisdictional diagram, which describes the organization of agents in terms of their authority status. It relates superior and subordinate agents.

The conversational model represents the interaction between agents. The purpose of the conversational model is to identify what messages are being exchanged in order for the agents to cooperate and negotiate with each other. The conversational model is derived from the agent relationship model and the internal agent model. The conversational model identifies what messages are being exchanged in the agent relationship model in order to fulfill the dependencies and detect the jurisdictional relationships.

The contract model defines a structure that captures commitments between agents. It specifies obligations and authorizations between the different agents about the services provided to each other. Contracts can be created when agents are instantiated or during execution, as and when they are needed. This model defines the expectations of how agents can fulfill the dependencies defined by the dependency model. The model also defines the expectations of agents when they play the roles defined by the jurisdictional model. Conversations are used as guidelines for discovering those expectations.

3.4.1.3 The PASSI Methodology

The PASSI methodology is an object oriented-based methodology. It was developed in 2002 by Cossentino and Potts. PASSI is composed of five models that address different design concerns and twelve steps in the process of building a model.

PASSI uses UML as the modeling language because it is widely accepted both in the academic and industrial worlds. Its extension mechanisms facilitate the customized representation of agent-oriented designs without requiring a completely new language. Extension mechanisms here refer to constraints, tagged values and stereotypes. The models and phases of PASSI are (see figure 3.4):

System requirements model is an anthropomorphic model of the system requirements in terms of agency and purpose. Developing this model involves four steps:

1. **Domain Description:** is a functional description of the system composed of a hierarchical series of use case diagrams. Scenarios of the detailed use case diagrams are then explained using sequence diagrams.
2. **Agent Identification:** The separation of responsibility into agents, represented as stereotypical UML packages. In this step, one or more use cases are grouped into stereotyped packages to form agent identification diagram.
3. **Role Identification:** The use of sequence diagrams to explore each agent's responsibilities through role-specific scenarios.
4. **Task Specification:** Specification through activity diagrams of the capabilities of each agent.

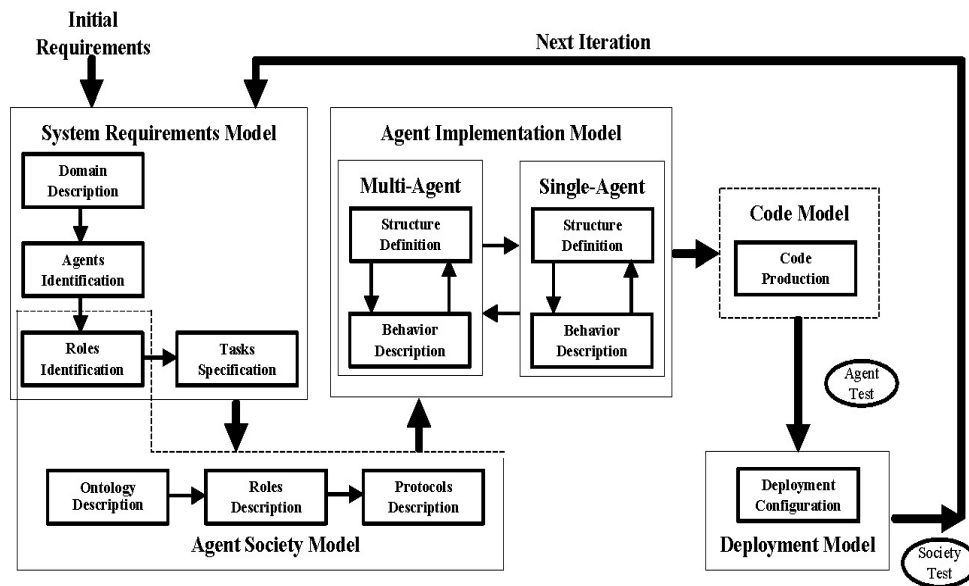


Figure 3.4 Models and phases of the PASSI methodology

Agent society model is a model of the social interactions and dependencies among the agents involved in the solution. Developing this model involves three steps in addition to a part of the previous model:

1. Role Identification. See the System Requirements Model.
2. Ontology Description: The use of class diagrams and Object Constraint Language (OCL) constraints to describe the knowledge ascribed to individual agents and the pragmatics of their interactions.
3. Role Description: The use of class diagrams to show distinct roles played by agents, the tasks involved that the roles involve, communication capabilities and inter-agent dependencies.
4. Protocol Description: The use of sequence diagrams to specify the grammar of each pragmatic communication protocol in terms of speech-act performatives like in the AUML approach [Odell 2001].

Agent implementation model is a model of the solution architecture in terms of classes and methods, the development of which involves the following steps:

1. Agent Structure Definition: The use of conventional class diagrams to describe the structure of solution agent classes.
2. Agent Behaviour Description: The use of activity diagrams or state charts to describe the behaviour of individual agents.

Code model is a model of the solution at the code level requiring the following steps to produce:

1. Code Reuse Library: A library of class and activity diagrams with an associated reusable code.
2. Code Completion Baseline: The source code of the target system.

Deployment model is a model of the distribution of the parts of the system across hardware processing units and their migration between processing units. It involves one step:

1. **Deployment Configuration:** The use of deployment diagrams to describe the allocation of agents to the available processing units and any constraints on migration and mobility.

Testing: the testing process has been subdivided into two different steps:

1. The (single) agent test is devoted to verifying its behaviour concerning the original requirements of the system solved by the specific agent.
2. The society test is used for the validation of the correct interaction of the agents, in order to verify that they concur in solving problems that need cooperation.

3.4.1.4 MaSE Methodology

The Multi-agent Systems Engineering (MaSE) [DeLoach 2004] methodology is considered as an object oriented-based approach. It provides a complete-lifecycle methodology to assist system developers to design and develop a multi-agent system. It describes the process, which leads a system developer from an initial system specification to system implementation. This process consists of seven steps, divided into two phases. Figure 3.5 illustrates the process of MaSE methodology.

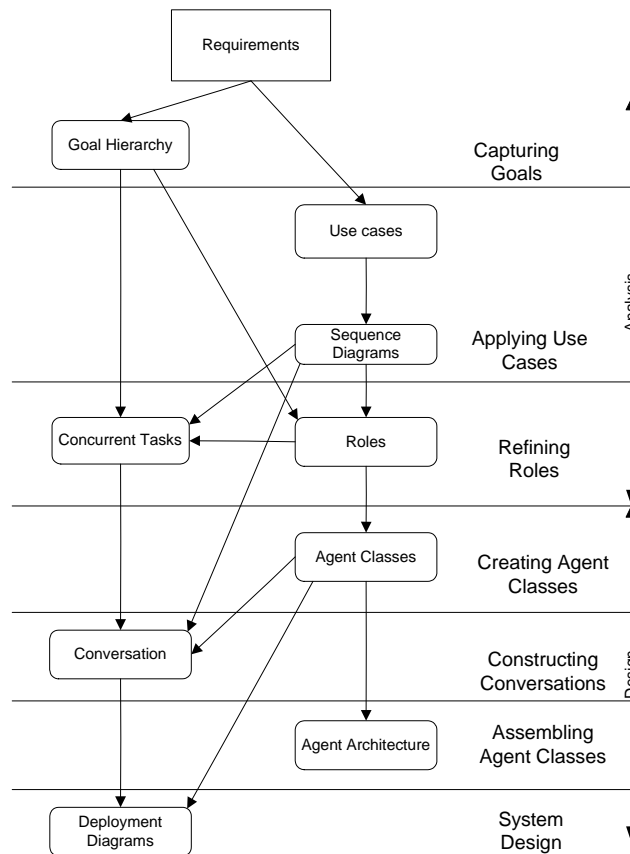


Figure 3.5 MaSE models and phases [DeLoach 2004]

The MaSE **analysis phase** is composed of three smaller process steps. The first step is capturing goals, which guides the developers to identify goals and then structure and represent them as a **goal hierarchy**. The second step, applying **use cases**, involves extracting main scenarios from the initial system context or copying them from it if they exist. These use cases are also used to build a set of **sequence diagrams** (similar to UML sequence diagrams). The final step is refining roles where a role model and a concurrent task model are constructed. The **role model** describes

the roles in the system. It also depicts the goals, which those roles are responsible for, the tasks that each role performs to achieve its goals, and the communication path between the roles. Tasks are then graphically represented in fine-grained detail as a series of finite machine automata in the **concurrent task model**.

The MaSE **design phase**, the first step of the design phase is creating agent classes. The result of this step is an **agent class diagram**, which describes the entire multi-agent system. The agent class diagram shows agents and the roles they play. Links between agents show conversations and are labeled with the conversation name. The details of the conversations are described in the second step of the design phase constructing conversations using **communication class diagrams**. These conversations are represented by a finite state machine. The third step of the design phase is assembling agent classes. During this step, we need to define the agent architecture and the components that build up the architecture. In terms of **agent architecture**, MaSE does not dictate any particular implementation platform. The fourth and final step of the design phase is system design. It involves building a **deployment diagram** that specifies the locations of agents within a system.

MaSE has an extensive tool support in the form of agentTool [DeLoach 2004]. The latest version of agentTool implements all seven steps of MaSE and provides automated support for transforming analysis models into design constructs.

3.4.1.5 MAS-CommonKADS Methodology

MAS-CommonKADS is one of the methodologies that are based on the knowledge engineering-based approach. This methodology [Iglesias, Garrijo, Gonzalez and Velasco 1999] is considered an extension of the CommonKADS methodology [Glaser 1997], as techniques from object-oriented methodologies (OOSE, OMT) and from protocol engineering were added for describing the agent protocols. It consists of three main phases: conceptualization, analysis, and design. These phases comprise of seven models that cover the main aspects in the development of multi-agent systems. Figure 3.6 illustrates the models of the MAScommonKADS methodology.

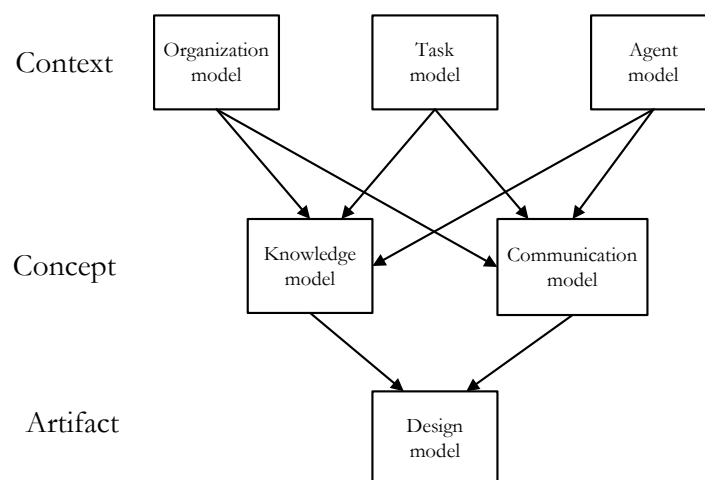


Figure 3.6 Models and Phases of the MAScommonKADS Methodology

The methodology starts with a **conceptualization phase**, which is an informal phase for collecting the user requirements and obtaining a first description of the system from the user's point of view. The use cases technique is used and the interactions of these use cases are then formalized with MSC (Message Sequence Charts).

The analysis and design phases define models as described below. For each model, the methodology defines the system components (constituents "entities to be modeled") and the relationships between these components. The methodology defines a textual template for describing every constituent and a set of activities for building every model. This is based on the development state of every constituent (empty, identified, described, or validated). These activities facilitate the management of the project.

The following models represent the extension of CommonKADS:

Agent model: The agent model specifies the characteristics of an agent including reasoning capabilities, skills (sensors/effectors), services, goals, etc. The agent model plays the role of a reference point for the other models. An agent is defined as any entity (human or software) capable of carrying out an activity. The identification of agents is based on the use cases diagrams generated in the conceptualization. Such identification could be augmented in the task model.

Task model: Describes the tasks (goals) that the agents can carry out. UML Activity diagrams are used to represent the activity flows and the textual template to describe the task (name, short description, input and output ingredients, task structure, etc).

Expertise model: Describes the knowledge needed by the agents to carry out the tasks. The knowledge structure follows the KADS approach. It distinguishes domain, task, inference and problem solving knowledge. Several instances of this model are developed for modeling the inferences on the domain, on the agent itself and on the rest of the agents.

Coordination model: Describes the conversations between agents. That is agents' interactions, protocols and required capabilities. The coordination model provides two milestones. The first milestone is concerned with identifying the conversations and the interactions. The second milestone is concerned with improving these conversations with more flexible protocols such as negotiation, identification of groups, and coalitions. The interactions are modeled using the formal description techniques MSC (Message Sequence Charts) and SDL (Specification and Description Language).

Organization model: Describes the organization in which the MASs are going to be introduced and the organization of the agent society. It illustrates the static or structural relationships between the agents. This model also describes the agent hierarchy, the relationship between the agents and their environment, and the agent society structure. A graphical notation based on OMT is used to express these relationships, adding a special symbol in order to distinguish between agents and objects.

Communication model: Several agents can be involved in a task. This model helps with modeling the communicative transactions between systems involved. These are often human-to-system and system-to-human communications.

Design model: The design model includes the design of relevant aspects of the agent network, selecting the most suitable agent architecture and the agent development platform. The design model assembles the agent, task, expertise, coordination, organization and the communication models. This assembled collection, is subdivided by the design model to generate three sub-models:

- *Application design:* composition or decomposition of the agents of the analysis, according to pragmatic criteria and selection of the most suitable agent architecture for each agent.
- *Architecture design:* designing of the relevant aspects of the agent network: required network and knowledge.
- *Platform design:* selects the agent development platform for each agent architecture.

3.5 Agent Methodologies Discussion

The weakness and limitations of these methodologies were already discussed in chapter 2 “Motivation and contribution”. As mentioned previously in section 3.4.1, these methodologies are classified according to the different approaches as agent-based, object oriented-based, and knowledge engineering-based. A number of these methodologies have been recommended for agent-oriented development. Yet, it is not easy to select a specific one in order to employ it or even to evaluate them. This is because they usually differ in their premises, covered phases, models, concepts and the supported multi-agent system properties. Therefore, a detailed discussion will be presented about their advantages, and difficulties.

3.5.1 Advantages of Agent Methodologies

Some of the existing agent-oriented methodologies are based on strong disciplined foundations; they possess some advantages as follows:

- 1) Some methodologies take into account the idea of a society of agents or the idea of an organization that provides a coherent conceptual infrastructure for analysis and design of multi-agent systems.
- 2) Some methodologies support both levels (micro and macro levels) to construct and develop agent systems.
- 3) A few methodologies support models derivation, where some models can be derived directly from others.
- 4) Some methodologies that are considered as an extension of the software engineering approach provide a solid base for the development of multi-agents systems.
- 5) Some methodologies use well-known techniques such as UML, which is particularly interesting. These techniques facilitate comprehension and communication between the various agents involved during software development [Sabas, Delisle and Badri 2002].
- 6) Some methodologies explicitly provide the cooperation between agents and the concepts used to describe the type of control. Some others are less clearly specified.

- 7) Some of these methodologies include an early requirements analysis phase, which assists the developers to understand an application by studying its organizational setting.
- 8) Some methodologies are becoming close to a complete methodology for multi-agent systems. They treat most development phases, and they treat both inter-agent and intra-agent perspectives.
- 9) The methodologies that constitute an extension of knowledge-based methods provide models that take into account the agents' internal states much better.
- 10) Some methodologies provide relatively elaborated support for reusable models, which is a valuable aspect for any methodology.

3.5.2 Difficulties of Agent Methodologies

The existing agent-oriented methodologies suffer from some difficulties which are the main reasons for a number of limitations emerge. Those difficulties prevent agent-oriented methodologies from being utilized and practiced in a wide manner. These difficulties are:

- 1) There is no existing agreement or accord on agent theory. Up until this point in time, no agent-oriented standards have been established and accepted as standard. No agent-oriented methodology will be able to spread unless the agent model is standardized. This standardization is refers to what characteristics define an agent, what types of architecture are available for agents, what agent organizations are possible, and what types of interactions there are between agents, etc.
- 2) None of these methodologies are used and none are exploited in a wide manner.
- 3) All research that examined and compared properties of these methodologies has suggested that none of them are completely suitable for industrial development of MAS.
- 4) There is no systematic approach to identify the components of MAS. Most current methodologies require the designers and developers to identify all agents of the system. Therefore, a designer experience is very important and is essential for producing a quality MAS. Designers should be trained beforehand to have the necessary skills for such projects.
- 5) Although, there are new languages for programming agent behavior, there are no adequate development tools for representing agent structure. Languages tend to focus mainly on particular agent architecture.
- 6) Selecting a suitable methodology to be followed for MAS development processes is not an easy task. Therefore, a precise methodology needs to be presented to guide the team of developers towards the achievement of objectives.
- 7) Comparing methodologies is often difficult. This difficulty arises from the fact that it is not easy to evaluate them because they usually differ in their premises, covered phases, models, concepts and the supported multi-agent system properties.
- 8) There is no agreement on what a methodology is and on what it should consist of.

3.6 Agent Programming Languages

The MAS literature provides a large number of different proposals for agent-oriented languages. These range from standard programming languages, to logic-based languages, and various hybrid approaches. A number of languages exist for developing

agents and multi-agent systems. It is impossible to explain these languages in-depth; as a result we give only a few examples.

3.6.1 Standard Programming Languages

Many agent systems are probably programmed in C/C++ and Java. In addition to these standard-programming languages, numerous prototype languages for implementing agent-based systems have been proposed to enable programmers to realize agent concepts in a better manner. There is no doubt that Java is the most used programming language for developing agents and multi-agent systems. Its advantages remain in large libraries of functions covering several domains like concurrency and security. Moreover, it is easy to insert new functions in Java. Even though Java is an oriented-programming language, it is interesting for developing agents and multi-agent systems.

3.6.2 Logic-based Languages

We first address the class of logic-based languages that are partially characterized by their strong formal nature, normally based on logic. Amongst the most well-known languages following the agent-oriented paradigm are AGENT-0 [Shoham 1993], Concurrent MetateM [Fisher 1995], AgentSpeak (L) [Rao 1996], 3APL [Hindriks, de Boer, van der Hoek and Meyer 1999], FLUX [Thielscher 2005], Minerva ([Leite 2003] and [Leite, Alferes and Pereira 2002]), Dali [Costantini and Tocchio 2002] and ResPect [Omicini and Denti 2001].

3.6.3 Hybrid Approaches

A variety of well-known agent languages combine features of standard and logic-based languages. In this section, we discuss agent-programming languages, which are logic-based. At the same time, some specific constructs allowing for the use of code implemented in some standard programming languages are provided, such as 3APL [Hindriks, de Boer, van der Hoek and Meyer 1999], Jason [Bordini Hübner et al. 2005], IMPACT [Subrahmanian et al. 2000], Go! [Clark and McCabe 2004] and AF-APL [Shoham 1993].

3.7 Agent Development Platforms and Frameworks

A lot of platforms and frameworks are available that support the process of agent-oriented software development. Most of them include some underlying platforms which implement the semantics of the agent programming language. Most of them are built by and integrated into Java. However, some frameworks exist that are not so strongly tied to a particular programming language. Instead, these frameworks are more concerned with providing support for aspects such as agent communication and coordination. Examples of such platforms and frameworks are TuCSoN [Omicini and Zambonelli 1999], ZEUS [ZEUS 1999], JADE [JADE 1999], Jadex [Braubach, Pokahr, and Lamersdorf 2004], LEAP [LEAP 2000], AgenTool [AgenTool 2000], JATLite [JATLite 2000], FIPA-OS [FIPA-OS 2000], MADKIT [MADKIT 1999], JACK [Busetta, Ronnquist, Hodgson and Lucas 1999], DESIRE [Brazier, Jonker, and Treur 1998] and Intelligent Agent Factory [2000]. In this section, we focus on such frameworks, having chosen TuCSoN, JADE, Jadex, and DESIRE as illustrative examples.

DESIRE (DEsign and Specification of Interacting REasoning components) is a compositional development framework for MAS. It is based on a notion of compositional architecture, and developed by Treur et al. [Brazier, Jonker, and Treur, 1998]. In this framework agent design is established upon the following main aspects: process composition, knowledge composition, and relations between knowledge and process composition. In this component-based agent approach, an agent's complex reasoning process is developed as an interaction between the components representing the sub-processes of the overall reasoning process [Brazier, Jonker, and Treur, 1998]. The reasoning process is structured as a number of reasoning components that interact with each other. Components may or may not be comprised of other components. Components that are not further decomposed are called primitive components. The function of the overall agent system is based on the functionality of these primitive components plus the composition relation that coordinates their interaction. Specification of a composition relation may involve, for example, the possibilities of information exchange among components and the control structure that activates the components. The DESIRE approach has been used for applications such as load balancing of electricity distribution and diagnostic systems. Further information and documentation of the tools supporting the development and implementation of multi-agent systems based on DESIRE is available at [Brazier, Dunin-Keplicz, Jennings and Treur 1997].

JADE (Java Agent DEvelopment Framework) is a Java framework for the development of distributed multi-agent applications. It represents an agent middleware providing a set of available and easy-to-use services and several graphical tools for debugging and testing. One of the major goals of the platform is to support interoperability by strictly adhering to the FIPA specifications concerning the platform architecture in addition to the communication infrastructure. Furthermore, JADE is flexible and can be adapted to be used on devices with limited resources such as PDAs and mobile phones. JADE has been widely used over the last years by many academic and industrial organizations ranging from tutorials for teaching support in agent-related University courses to Industrial prototyping [Bordini, Dastani, Dix, and El-Fallah 2005]. As an example, Whitstein has used JADE to construct an agent-based system for decision-making support in organ transplant centres [Calisti, Funk, Biellman, and Bugnon, 2004]. The JADE platform is open source software and can be obtained at [JADE 1999].

Jadex [Braubach, Pokahr, and Lamersdorf 2004] is a software framework for the construction of goal-oriented agents following the belief-desire-intention (BDI) model. The framework is realized as a rational agent layer that sits on top of a middleware agent infrastructure such as JADE. It supports agent development with well-established technologies such as Java and XML. The Jadex reasoning engine addresses traditional limitations of BDI systems by introducing new concepts, such as explicit goals and goal deliberation mechanisms (see, e.g., [Braubach, Pokahr, Moldt, and Lamersdorf 2005]). This makes results from goal-oriented analysis and design methods (e.g., KAOS and Tropos) more easily transferable to the implementation phase. Jadex has been used to build applications in different domains such as simulation, scheduling, and mobile computing. It has also been successfully used in several software engineering courses at the University of Hamburg. The Jadex system, developed at the Distributed Systems and Information Systems group at the University of Hamburg, is freely available under the LGPL license and can be

downloaded from [Braubach, Pokahr, and Lamersdorf 2004]. Besides the framework and additional development tools, the distribution contains an introductory tutorial, a user guide, and several illustrative example applications with source code.

TuCSoN (Tuple Centre Spread over the Network) is a coordination framework for multi-agent system based on a model and a related infrastructure, which provides general-purpose, programmable services to support agent communication and coordination [Omicini and Zambonelli 1999]. The model is based on **tuple centres** which are runtime programmable abstractions whose coordinating behavior can be dynamically specified with a logic-based language called ReSpecT [Omicin and Denti 2000]. Tuple centres are coordination tools, which reside in the agent cooperative working environment, shared and used collectively by the agents to support their coordination. The TuCSoN technology is open source software and completely based on Java. It is comprised of: a runtime platform to be installed on hosts to turn them into nodes of the infrastructure; a set of libraries (APIs) to enable agents access to the services; and a set of tools mostly to support the runtime inspection and control (monitoring, debugging) of tuple centres' state and coordinating behavior. At the heart of the TuCSoN technology is the tuProlog technology, a Prolog engine fully integrated with the Java environment, available also as a standalone library and environment. Currently, TuCSoN is used as one of the reference platforms for building agent-based systems in academic projects and thesis developed at the Engineering Faculties in Cesena and Bologna.

3.8 Chapter Summary

In this chapter, the literature concerning agents, agent architectures, multi-agent systems, agent-oriented methodologies, and agent development programming languages and platforms was reviewed. The field of agent-oriented methodologies was examined with the aim of establishing the characteristics of agent-based systems. An analysis of agent-oriented methodologies that gives a clearer picture of their application domain was presented including the advantages and difficulties they present.

CHAPTER FOUR

REQUIREMENTS FOR A COMPREHENSIVE AGENT-BASED SOFTWARE ENGINEERING METHODOLOGY

4.1 Introduction

This chapter describes the requirements for a new agent-based software engineering methodology. These requirements are discussed based on the following three categories: concepts, models, and process.

4.2 Requirements for a New Methodology

A number of methodologies were studied in detail and their characteristics were analyzed. We identified many advantages and disadvantages of different aspects of the methodologies under study. Based on these findings, we compiled the following list of requirements that, in our view, the work involved in this research needed to address. These requirements are classified into three types of categories that new methodology should comply with: concepts, modeling techniques, and processes.

4.2.1 Requirements on the level of Concepts

- 1) The new methodology should be based on robust concepts of agent system and MAS. Therefore, it should have a complete conceptual agent and MAS structure.
- 2) The methodology should rely on a plain, specific conceptual framework, which is responsible for specifying and linking the concepts during the different construction stages. This conceptual framework is considered as a foundation in the different phases of construction.
- 3) The methodology should also deal with the agent concept as a high-level abstraction, capable for modeling complex systems.
- 4) The new methodology should close the gap between the design models of the methodologies and the existing implementation languages.
- 5) The methodology should take into account the idea of a society of agents or the idea of an organization. Therefore, the methodology should use explicit organizational aspects like role, responsibilities, permissions, goals, plans, and tasks.
- 6) The new methodology should be able to model the mental aspects of agents such as beliefs, goals, and plans. Such aspects play a crucial role in determining how rational agents will act.
- 7) The methodology should be able to support existing agent architectures in order to specify how the agent can be decomposed into a set of component modules and how these modules should be made to interact.

- 8) The methodology should provide concepts which are used to specify and represent changes in the environment, e.g. events, incidents, etc. Therefore, it should include a trigger concept for agents to represent its autonomy and reactivity characteristics.

4.2.2 Requirements on the level of Models

- 1) The methodology should use modeling languages that are widely accepted both in the academic as well as in the industrial world.
- 2) The methodology should utilize well-known techniques for requirement gathering and agent communication in order to link them to domain analysis and design models.
- 3) The design models in the new methodology should be easy to implement. This means that this methodology should have no complexity and it should have design constructs that can be mapped onto instructions of an available programming language.
- 4) The methodology should provide support for some essential software engineering issues that will have a substantial effect on its acceptability for industry and, thus on the adoption of the agent technology. Examples are: preciseness, accessibility, expressiveness, domain applicability, modularity, refinement, model derivation, traceability, and clear definitions.
- 5) The methodology should provide models to represent both the visualization of the agents' behavior and the definition of how this behavior is achieved.
- 6) The methodology and its models should be able to capture effectively the complexity of agent systems, agents' internal structure, relationships, conversations, and commitments. In addition, it should properly capture the behavior of agents, interactions between agents, and organizational structures.
- 7) The methodology should support models derivation, where some models can be derived directly from others. It should allow extracting a model from another with ease.
- 8) The methodology should define general analysis, and design models that can systematically perform an agent identification process.
- 9) The new methodology should have models that are easy to use (with understandable notations), easy to construct, easy to apply, easy to represent, and easy to trace through.
- 10) The methodology should have models with a notation capable of expressing models of both static aspects of the system and dynamic aspects.
- 11) The methodology should have models that do not contradict each other.

4.2.3 Requirements on the level of Process

- 1) The methodology process should have the attributes of simplicity, and ease of use as well as traceability.
- 2) This methodology should cover in sufficient depth all the following aspects: a full lifecycle process; a comprehensive set of concepts and models; a full set of techniques such as rules, guidelines, and a modeling language.

- 3) The methodology should have a reliable systematic approach that proves a milestone for Software Development Life Cycle (SDLC). It should be able to create a multi-agent system starting with the initial specification, system requirements and then producing a set of implementation codes.
- 4) The methodology process should include an early system requirements phase which provides a clear understanding and description of how the whole system works.
- 5) The methodology process should enable designers to clearly structure and construct the application as MAS.
- 6) The methodology should contain robust and general tools that are flexible enough to specify and implement the characteristics of the agents involved.
- 7) The methodology should be complete and cover the entire area from analysis to implementation rather than address different properties of software agents and methodological aspects.
- 8) The methodology should provide facilities to allow an agent to be capable to satisfy its needs, make use of its interests, and take control of its beliefs.
- 9) The methodology should have implementation constructs that have exact semantics.
- 10) The methodology process should provide an implementation language with explanations of how to implement the reasoning of beliefs, reasoning of goals and plans, and reasoning of communication.
- 11) The methodology process should provide mechanisms to specify and represent agents' responses to changes in the environment.
- 12) The methodology should support describing an agent's self-control features.
- 13) The methodology should be able to support goal-modeling techniques, which capture the agents' goals. It should permit to model plans and their tasks, which describe how an agent achieves goals.
- 14) The methodology process should provide the agent's ability to cooperate with other agents.
- 15) The methodology should allow for agents modeled in the methodology to be able to store information about their environment and their internal states as well as the actions they may carry out.
- 16) The methodology should support modeling dependencies between agents. In addition, should allow agents modeled in the methodology to depend on each other to achieve goals.
- 17) The methodology should support both asynchronous and synchronous communication modes between agents.
- 18) The methodology should have a way of representing the communication protocols.
- 19) The methodology should support easy illustrating and testing.

PART TWO

SOLUTION

The second part of this research work starts by discussing the entire process of the new methodology in detail. The process of the new methodology is described step by step in order to explain how the structure of the new methodology works to build agent systems. This is described in chapter 5.

This part also describes the complete detailed process of developing multi-agent systems by using a case study of the car rental system. This case study is used to prove the methodology. This described in chapter 6.

CHAPTER FIVE

MULTI-AGENT SYSTEM DEVELOPMENT METHODOLOGY

5.1 Introduction

This chapter presents in detail the new Multi-Agent System Development (MASD) methodology. The chapter starts with the assumptions of the new methodology. It then states the methodology construction necessities and then describes the phases of the proposed methodology. The MASD methodology consists of four phases: the system requirement phase, analysis phase, design phase and implementation phase. Each phase is described in detail.

5.2 Assumptions

Before describing the development process of the new methodology, we provide a few limitations or restrictions in order to define the scope of the new MASD methodology.

The first restriction that the current version of the MASD methodology is designed to work for cross-boundary systems (semi-open systems) where the agent society itself is closed (i.e. the types and behaviours of agents defined in the system are determined beforehand) but external agents may interact with members of the society via the defined and used protocols (e.g., FIPA).

The second restriction is that the current version of MASD methodology is focused on small and medium sized systems. We assume up to fifteen agents. This number is not a hard limit, but simply no verification is done for larger systems. A large number of agents may lead to complex inter-agent communications

The third restriction is that our MASD methodology is based on the BDI agent architecture, which is used to design agents for the development process. Moreover, this methodology follows the traditional top-down approach that starts by identifying the system requirements and ends up by implementing the system.

The fourth restriction is that there is no requirement for agent mobility. A mobile agent is one that can move between computers hosting the MAS. One method of accomplishing this mobility is for the agent to start a new version of itself at another site, send it the state information from the old version, then terminate, and delete the old version. This creates a new copy of the agent that continues where the old one left off. This produces several problems for the rest of the system such as updating all other agents with the new location of the mobile agent. In addition, what happens if an agent transfer goes wrong and there are multiple or no copies of an agent? The inclusion of mobile agents may add more complexity to the methodology, however does not add much to its functionality. Most of the benefits of mobile agents can be designed into a system by simply using multiple agents.

The fifth restriction is that MASD does not consider dynamic systems where agents can be created and destroyed during execution. This would lead to many of the problems as with mobile agents. An agent can be added to a system through a process such as registration, which is a user-initiated event, but cannot be added and deleted continuously during ordinary operation.

The sixth restriction is that the MASD methodology does not support any models, which can be used to assess whether a multi-agent approach is suitable. MASD assumed that the system is already suitable to be developed as MAS.

The final restriction is that inter-agent conversations are assumed to be one-to-one, only as opposed to multicast. This assumption was made after an investigation of conversation representation, and acceptance of a graphical dual-state table representation. Substituting a series of point-to-point messages will fulfill a requirement for a multicast message.

5.3 MASD Methodology

MASD methodology is developed as a reliable systematic approach that proves a milestone for Software Development Life Cycle (SDLC). Figure 5.1 illustrates the process of MASD Methodology. The proposed methodology covers the most important characteristics of multi-agent systems. The new methodology deals with the agent concept as a high-level abstraction capable of modeling a complex system.

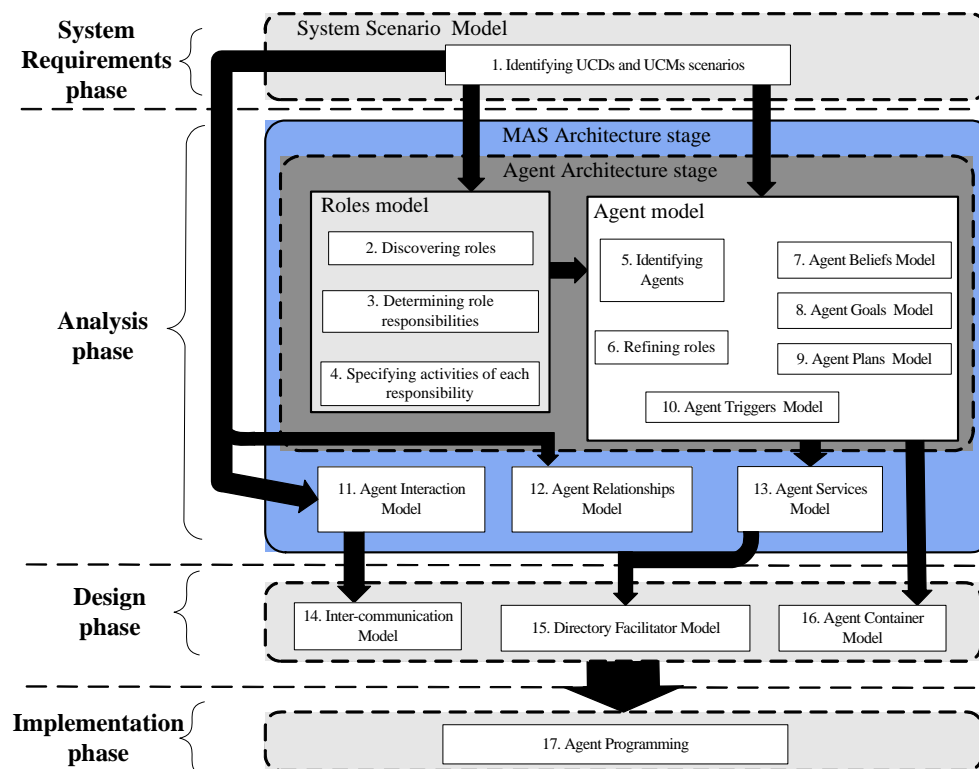


Figure 5.1 MASD Methodology process

In addition, it includes well-known techniques for requirement gathering and customer communication and links them to domain analysis and design models such as UCMs [Buhr 1998], UML Use Case Diagrams [UML Use case diagrams], Activity

diagrams [UML Specification 1997], FIPA-ACL [FIPA], etc. Furthermore, it supports simplicity and ease of use as well as traceability.

The MASD methodology is composed of four main phases; the system requirements phase, the analysis phase, the design phase, and the implementation phase. The next sections present a more discussion of each of the four phases. A car rental system [EU-Rent] is used to describe the process of MASD methodology. In this chapter, we use only the reservation scenario as an example. The full system scenario will be described in detail later by a case study in chapter 6.

5.3.1 System Requirements Phase

The system requirements phase describes all the details of a system scenario as a high-level design through system scenario model. The system scenario model uses well-known techniques such as Use-Cases Diagrams (UCDs) and Use Case Maps (UCMs) [Buhr 1998] to describe the whole system scenario. Such techniques assist to discover the system components such as agents, objects, roles, resources etc. and their high-level behaviour. The system requirements phase produces a model called the **system scenario model**.

5.3.1.1 System Scenario Model

This model is used as a starting point for generating more detailed visual descriptions. It describes the whole system scenario in terms of what a system does, but it does not specify how it does it. The model captures the components that the system is composed of and the tasks that have to be performed by each component within the system. Then, it illustrates how these components interact with each other and with the external environment. In addition, it captures the behavior of a system as it appears from the point of view of the outside user. To construct this model, some specific, well-known techniques have been used such as Use-Case Diagrams (UCDs) and Use Case Maps (UCMs). These techniques are assembled together in order to understand and obtain a complete system requirement as far as possible.

In the system scenario model, UCDs are exploited to describe the behavior of the system from the user's point of view. It is through this notation that the roles in the system can be recognized. Recognition of roles within a system is very helpful during the analysis and design phases as well as for understanding the system's requirements. More detail explanations about UCMs are found in Appendix B.

Also, in the system scenario model, UCMs are used as a precise structured notation. UCMs describe system scenario in terms of causal relationships between responsibilities. They also emphasize the most relevant, interesting and critical functionalities of the system. They describe the general behavior of the system in the form of scenarios without referring to any implementation details. More detailed explanations about UCMs are found in Appendix A. UCMs include adequate information in a summarized form. It has two advantages:

- It enables developers to understand and conceptualize the behaviour of the system as a whole.
- It gives an explicit concept overview about how the system operates as a whole.

5.3.1.2 Integrating UCMs and UCDs

At the system requirements phase, the system scenario model is developed by constructing the UCDs as well as UCMs. Before starting to build this model, it is necessary to explain why these techniques were chosen and what is the importance of integrating them together in this model.

There are several alternative techniques used to capture user requirements, such as IEEE standard Software Requirements Specification (SRS) STD 830-1998, UML-UCDs and UCMs etc. We have chosen UML-UCDs and UCMs techniques to describe the system requirements phase because these techniques are comparatively more appropriate than others for agent-based systems [Unland, Abdelaziz and Elammari 2004]. Moreover, their components can be easily transferred or mapped to agent concepts and models. In addition, they have the following important advantages [Amyot¹ 2001] and [Amyot² 2001]:

- Advantages in the description of interactive systems.
- Advantages in the description of the dynamic behaviour.
- Advantages in the description of aspects of the system's behaviour in the form of simple and clear scenarios.
- The ability to handle complex distributed systems and the ability to describe them in a high-level view in a flexible and concise manner, without referring to the details of implementation.
- The ability to bridge the gap between requirements and design.

It is also worth pointing out the main reasons behind integrating these diagrams together and the advantages of their presence together in a system requirement phase.

The first reason is that UML-UCDs are integrated with UCMs in order to provide a powerful concept for visualizing how the system works as a whole. UCDs and UCMs together provide a good description for common communication between project members. UCDs explain preconditions, postconditions, and critical scenarios. UCMs provide a visual notation for those use cases and a means of extending them into high-level design [Rys 2005].

The second reason is that a number of agent-oriented methodologies utilize UCMs and UCDs techniques in the system requirements description phase. Two methodologies that use UCMs are HLIM [Elammari and Lalonde 1999] and Styx [Bush 2001]. Methodologies that use UCDs are AUML [Odell 2001], MaSE [DeLoach 2004] and PASSI [Cossentino and Potts 2002]. Most of these methodologies still suffer from the problem of incompleteness of the system requirements description phase. Such methodologies fail in some extent to obtain a sufficient description of the requirements [Amyot² 2001]. That is because there is a conceptual gap existing between the functional requirements (UCDs) and their realization in terms of behavioral diagrams (the design) [Rys 2005] and [Amyot¹ 2001].

Figure 5.2 illustrates the existing mentioned gap between the functional requirements and the design. UCMs are a scenario-based software engineering technique most useful at the early stages of software development. UCMs represent a complementary part to bridge the gap between requirements and design by combining behavior and structure in one view and by flexibly allocating scenario responsibilities

to architectural components. Requirements and use-cases usually present a view that describes the system according to its external behavior. UML behavioral diagrams provide a view that describes the internal behavior in a detailed way. UCMs can provide a traceable progression from functional requirements to detailed views based on components and interactions, while at the same time combining behavior and structure in an explicit and visual way.

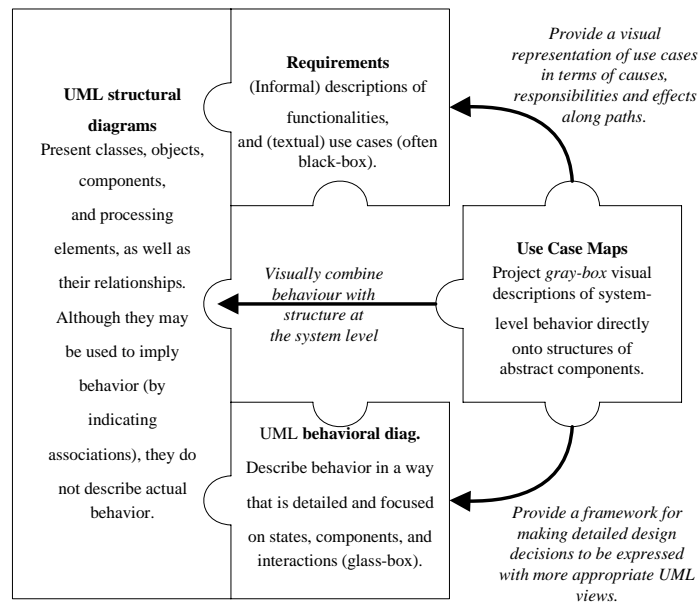


Figure 5.2 The gap between the functional requirements and the design [Amyot¹ 2001]

The third reason is that these techniques are considered complementary to each other. UCDs are part of the UML tools in describing interactions between the user and the system. They play a large role during the system's building stages. In addition, they assist in understanding the requirements. Moreover, UCMs visually combine behavior with structure at the system level. UCMs are used as a visual notation for describing causal relationships between responsibilities of one or more use cases in UCDs in terms of sequences of responsibilities along paths. The notation is applicable to use case capturing and elicitation, use case validation, as well as high-level architectural design and test case generation [Buhr and Casselman 1996].

Furthermore, UCMs can be transferred and can be mapped to other architectures consisting of different types of components, such as MSCs [Bordeleau 1997], FSM [Bordeleau 1999], and SDL [Sales 2000]. They also support the development of complex structured scenarios on a high-level of abstraction, in addition to their integration with each other. They also have the ability to identify and describe changes of system behavior during run-time. It describes in an organized and specific manner the structures of system scenario through sub-diagrams, developed and drawn through symbols called stubs.

In fact, this is the motivation that encouraged us to integrate these techniques in one model called the system scenario model in the system requirements description phase. Thus it is possible to benefit from the advantages of each technique individually as well as to obtain a comprehensive and complete description of the system requirements. Furthermore, the integration of these diagrams provides a high

level of full understanding and comprehensive description of all the desired system requirements.

The system scenario model consists of two steps. The first step is to develop the use case diagrams. The second is to construct the use case maps. This model is realized by the following four points:

- The use of UCDs, which in turn describe interactions that take place between the user and the system. It describes the behavior of the system from the user's point of view. It also assists in understanding the requirements.
- The use of UCMs diagrams to provide a high-level view that describes the general behavior of the system as a whole in the form of a scenario without referring to any details regarding the implementation of the system. UCMs provide precise structural symbols, which contain enough information in a concise form to enable individuals to understand, conceive, and visualize the behavior of the system and to form a clear idea about how the entire system works.
- The use of UCMs as a visual notation to describe causal relationships between responsibilities of one or more use cases in UCDs in terms of sequences of responsibilities along paths.
- The use of UCMs assists to capture and elicit use cases and validate them.

5.3.1.3 Reservation Scenario Example

Before starting to describe the system scenario model, we have to introduce a brief description of the reservation scenario. Most rentals are by advance reservation. The rental period and the car group are specified at the time of reservation. Reservation can be achieved online by filling web application, or can be achieved by sending an e-mail, or by a phone call.

5.3.1.3.1 UCD of Reservation Scenario

In this section, a detailed example will be provided which will perform the construction of UCDs for reservation scenario of the EU-Rent a car Rental Company. Each use case in the reservation scenario will be described with a diagram as well as describing and clarifying its components. Initially, use case diagrams of the dialogues for the reservation scenario will be created as in figure 5.4. It will be followed by a description of each use case separately.

In order to develop UCDs for a reservation scenario, we should capture the following system components: the actors involved in the reservation scenario and the use cases performed by those actors. Figure 5.3 shows use case diagram notations.

In each use case, we should perform the following tasks:

1. Identify the description of use case.
2. Identify the actor that performs the use case.
3. Identify the goal of the use case.
4. Identify preconditions and postconditions of each use case.
5. Identify triggering events of the use case.
6. Identify extensions and alternatives to the use case.

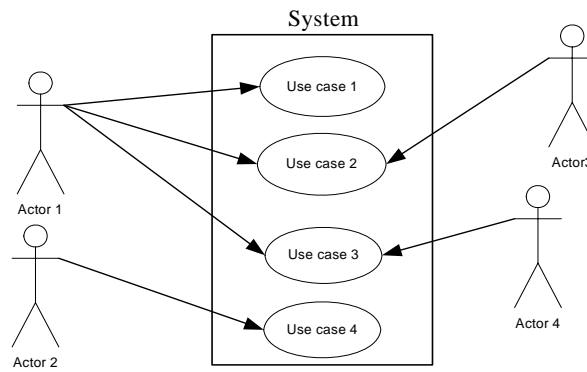


Figure 5.3 Use Case Diagram notations

The following UCD describes the reservation scenario. In this scenario, there are two actors: The customer actor and the car rental clerk actor. Each actor represents a role that a user system plays within the system. Actors can be a human or an automated system. A use case is made up of a set of scenarios. Each scenario is a sequence of steps that encompasses an interaction between a user and a system. The customer actor requests the car rental clerk actor to perform an action, such as reserve a car or cancel a reservation. The clerk actor performs actions such as replying to customer requests and so on. Figure 5.4 illustrates a UCD for a reservation scenario.

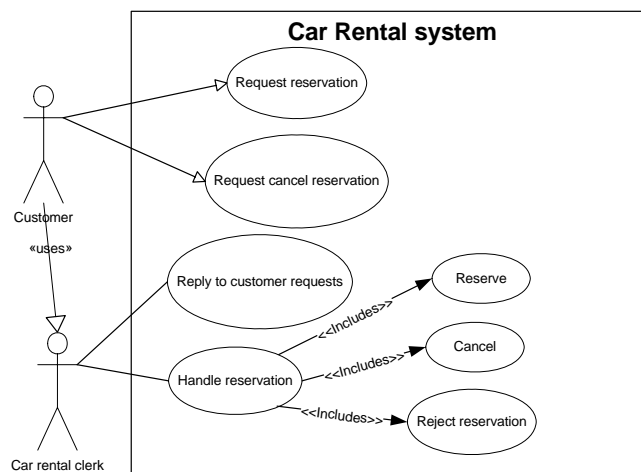


Figure 5.4 Use Case Diagrams for car rental system

We used the semi colon symbol “;” in preconditions, postconditions and triggering events to represent the “or” operator. We used the comma symbol to “,” in preconditions, postconditions and triggering events to represent the “and” operator.

Use case **Request reservation**

Use case name: Request reservation

Description: The customer requests the car rental clerk for reservation

Actors: Customer

Goal: To request car rental reservation

Precondition: Customer requested reservation

Postcondition: Request rejected; Request accepted

Triggering event: A customer requests a reservation

Extensions:

Alternatives:

Use case **Request cancel reservation**

Use case name: Request cancels reservation

Description: The customer requests the car rental clerk for canceling reservation

Actors: Customer

Goal: To request cancel reservation

Precondition: Customer has reservation and requested cancellation

Postcondition: Request rejected; Request accepted

Triggering event: a customer requests cancel reservation

Extensions:

Alternatives:

Use case **Reply to customer requests**

Use case name: Reply to customer requests

Description: The Car rental clerk replies to customer requests

Actors: Car rental clerk

Goal: To provide services to the customer

Precondition: Customer requests a specific service

Postcondition: The customer replied to

Triggering event: A customer requests a reservation; customer requests a cancellation;
customer requests extending the rentals.

Extensions:

Alternatives:

Use case **Handle reservation**

Use case name: Handle reservation

Description: The car rental clerk handles the reservation that takes place

Actors: Car rental clerk

Goal: To achieve the reservation process

Precondition: The reservation requested by the customer; the cancellation requested by the customer

Postcondition: The reservation request dealt with

Triggering event: The customer requests reservation

Extensions: Allocate car for customer, Reserve, Cancel, or Reject reservation

Alternatives:

Use case **Reserve**

Use case name: Reserve

Description: The car rental clerk reserves a car of a specific category for a specific customer

Actors: Car rental clerk

Goal: To reserve a specific car

Precondition: Customer requested a reservation

Postcondition: A new reservation exists; specific car is reserved for the customer

Triggering event: The customer requests the clerk to reserve a car for him/her.

Extensions: - Check blacklist, verify rules and check customer demands

Alternatives:

Use-case **Cancel**

Use case name: Cancel

Description: The car rental clerk cancels a reservation that already exists

Actors: Car rental clerk

Goal: To prevent the picking up of a car for which a reservation was made

Precondition: The reservation exists

Postcondition: The reservation is marked as cancelled; no car will be picked up for this reservation

Triggering event: A customer requests the clerk to cancel a reservation

Extensions:

Alternatives:

Use case **Reject reservation**

Use case name: Reject reservation

Description: The car rental clerk rejects the reservation

Actors: Car rental clerk

Goal: To prevent rentals which are against car rental rules

Precondition: The customer request is made; the customer does not fit the car rental rules
Postcondition: The reservation is rejected
Triggering event: Reservation rules not satisfied
Extensions:
Alternatives:

5.3.1.3.2 UCMs of Reservation Scenario

In this model, use case maps for reservation scenarios of the car rental system are described. This section describes how UCMs can be used to represent and describe the reservation scenario of the car rental system. UCMs are applied to capture the behavior of the system as high-level description and explain how UCMs describe the reservation scenario in visual views. The following scenarios represent interactions between some components in the system. By tracing application scenarios, the high-level view of the system is derived. These scenarios describe functional behavior as UCM paths within the system. This discovers system roles, responsibilities, and plugins along the way. UCMs perform the most important steps:

1. Identify scenarios and major components involved in the system.
2. Identify roles for each component.
3. Identify preconditions and postconditions for each scenario.
4. Identify responsibilities and constraints for each component in a scenario.
5. Identify sub scenarios and replace them with stubs.
6. Identify components collaborations for the major tasks.

In order to develop UCMs for reservation scenario we have to introduce UCMs notations, which are described briefly in figure 5.5. UCMs are described in more detail in Appendix A.

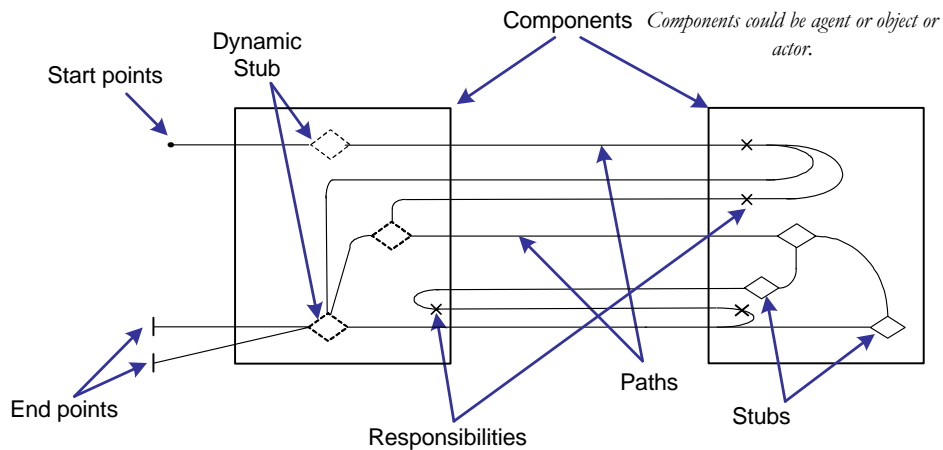


Figure 5.5 Use Case Maps Notations

The reservation scenario will be performed between two components of the system called: customer and car rental clerk. Figure 5.6 shows the use case map for the reservation scenario.

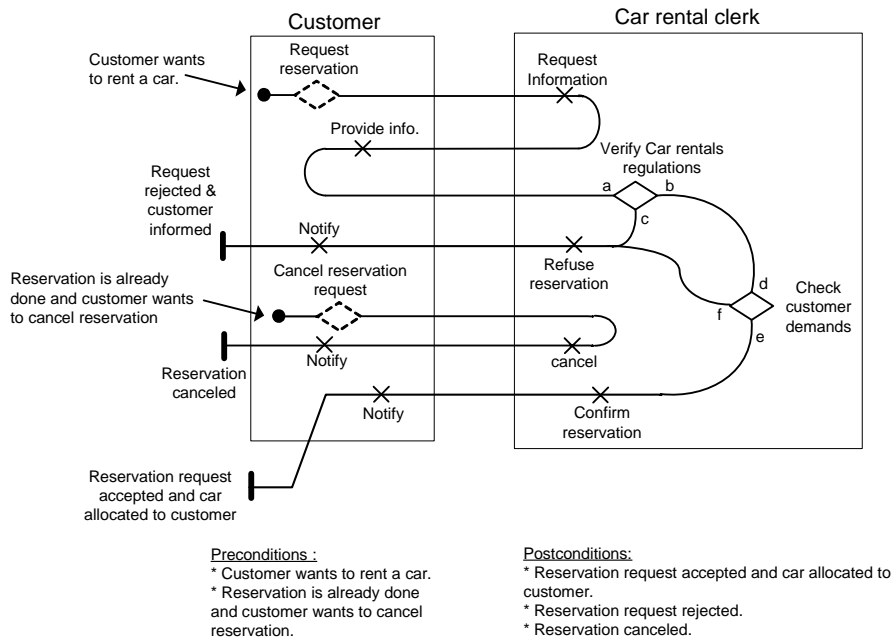


Figure 5.6 Use Case Map for Reservation Scenario

The customer component represents the customer in the application environment, and the car rental clerk represents the employee of the car rental company. The preconditions for the reservation scenario are:

- A customer wants to rent a car.
- A reservation is already done and the customer wants to cancel the reservation.

When the first precondition is satisfied the scenario starts with the *request reservation* stub, which hides the detailed information of the *request reservation* process. The request reservation can be achieved in several ways. For example, it can be done by a phone call, or by filling a web form, or by an Email. Therefore, the *request reservation* stub is represented as a dynamic stub. Figure 5.7 illustrates the plug-ins for the *request reservation* dynamic stub. After all responsibilities for the *request reservation* process are performed, the path leads to the car rental clerk component. In this component there is a responsibility called *request information*, which requests the customer to provide his/her personal information such as address, phone, personal ID, driving license etc. The path leads to the customer component where there is a responsibility called *provide info*, which provides a confirmation that the customer has filled in the application form for the rental transaction.

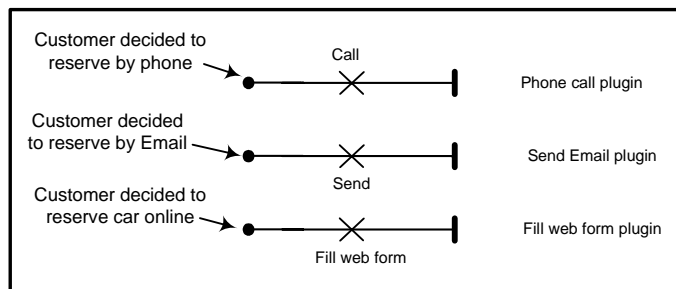


Figure 5.7 Plug-Ins for Request Reservation Stub

After the previous responsibilities are performed, the path leads to the static stub *verify car rentals regulations* which hides the detailed information of the *verify car rentals regulations* process. This stub should be achieved in one specific mode.

Figure 5.8 illustrates the plug-in for the *verify car rentals regulations*. In this plug-in the car rental clerk checks whether the customer meets the rental rules of the car rental company.

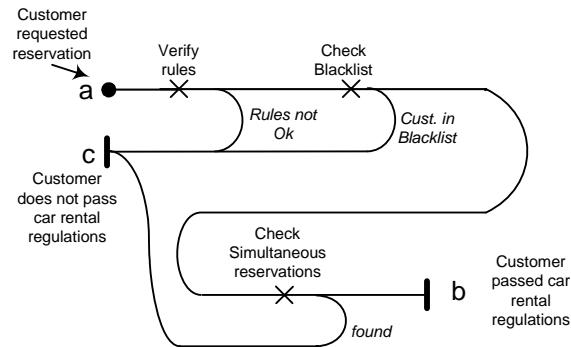


Figure 5.8 Plug-In for Verify Car Rentals Regulations

These regulations are represented by the following tasks or responsibilities (*verify rules*, *check blacklist* and *check simultaneous reservations*) that should be performed by the clerk of the car rentals company. The path starts with the *verify rules* responsibility which verifies the rules of the car rentals company such as customer age, validity of drivers license etc. Then the path leads to an or-fork immediately after the *verify rules* responsibility, which indicates alternative scenario paths.

One path leads to the end point “c” which means the *reservation request* is rejected, e.g. because the customer does not have a valid drivers license. The other path leads to the next responsibility *check blacklist*. The *check blacklist* responsibility checks whether the customer belongs to the customers blacklist or not. In the same situation, the path leads to an or-fork, which immediately indicates alternative scenario paths. One path leads to the end point “c” which means the reservation request is rejected, since the customer is included in the blacklist. The other path confirms that the customer is not included in the blacklist, leading to the last responsibility *check simultaneous reservations*. It checks whether the customer has reservation for more than one car at a time. A customer may have multiple future reservations, but may have only one car at any time. After the *check simultaneous reservations* responsibility is checked the path leads immediately to an or-fork, which indicates alternative scenario paths. One path leads to the end point “c” which means the reservation request is rejected, since the customer already has another car, which according to the car rentals rules is not allowed. The other path leads to the end point “b” which confirms that customer passed the car rentals regulations and he/she is allowed to reserve a car.

The *verify car rentals regulations* stub has two outgoing ports. If the customer passed the car rentals regulations, port “b” will be followed, which means that the customer is allowed to reserve a car. Otherwise, port “c” is followed, which means that the customer reservation request is rejected. The path that comes from port “b” leads to the *check customer demands* stub, which hides the detailed information of the *check customer demands* process. This stub checks whether the customer demands are available or not.

Figure 5.9 illustrates the plug-ins for the *check customer demands* stub. In this plug-in, the car rental clerk checks the customer's demands.

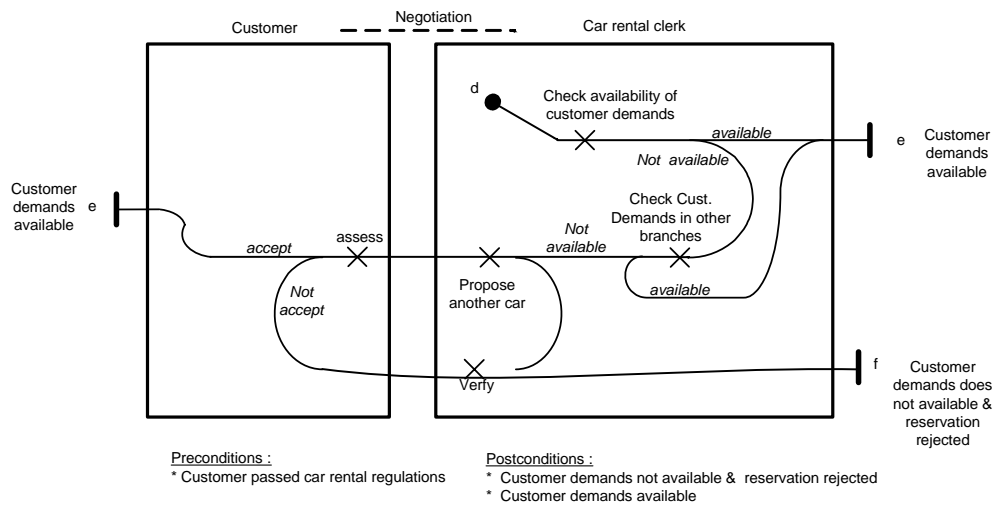


Figure 5.9 Plug-In for the Check Customer Demands Stub

This plug-in is represented by the following tasks or responsibilities: check availability of customer demands, check customer demands in other branches, propose another car, assess, and verify. The path is started with the responsibility *check availability of customer demands* which checks whether the customer's demands are available in this branch or not.

After that, *check availability of customer demands* responsibility invoked, the path leads to an or-fork, which indicates alternative scenario paths. One path labeled *available* leads to the end point "e" which means the customer demands are available. The other path (labeled *Not Available*) leads to the responsibility *check customer demands in other branches*. This responsibility finds out whether the customer demands are available in other branches or not.

After *check customer demands in other branches* responsibility is checked, the path immediately leads to an or-fork, which indicates alternative scenario paths. One path labeled *available* leads to the end point "e" which means the customer's demands are available in some other branch. The other path (labeled *not available*) leads to responsibility *propose* which proposes to the customer another car from the same group. The path after that leads to the responsibility *assess*, which confirms that the customer estimates the proposal. Then the path leads to an or-fork. One path labeled *accept* that leads to the end point "e".

That is an indication that the customer accepts the proposal. The other path (labeled *not accepted*) leads to the responsibility *verify* which verifies that the customer has responded. If the customer asks for another offer the path leads to the responsibility *propose* again. Otherwise, the path leads to the end point "f" which means that the customer demands were not available and the customer reservation is rejected.

The *check customer demands* stub should be returned back to the reservation scenario either by the “e” or “F” port. The path that comes from port “e” leads to the responsibility *confirm reservation* and then leads to the end point *reservation request accepted and car allocated to customer*. The path that comes from the port “F” leads to the responsibility *refuse reservation* and then the path leads to the end point *request rejected and customer informed*.

When the second precondition of the reservation scenario is satisfied; the scenario starts with the *cancel reservation request* stub, which hides the detailed information of the *cancel reservation request* process. The *cancel reservation request* stub can be achieved in several ways. For example, it can be done through a phone call, by filling a web form, or by an Email. Figure 5.10 illustrates the plug-ins for the *cancel reservation request* stub. After all responsibilities for the *cancel reservation request* process are performed, the path leads to the car rental clerk component where there is a responsibility called *cancel*, which cancels the reservation that is already done by the customer. Then the path leads to a responsibility *confirm* which confirms that the reservation is canceled. Then the path leads to the customer component where there is a responsibility called *receive confirmation*, which indicates that the confirmation for cancellation is received by the customer. Then the path leads to the end point *reservation canceled*.

At the end of this phase, the general behavior of the system as a whole is described in a high-level view using UCMs scenarios. The interactions that take place between the customer and the car rental system are described and the system requirements are understood using UCDs and UCMs. UCMs and UCDs describe the system without referring to any details regarding the implementation of the system. They provide a clear idea about how the entire system works.

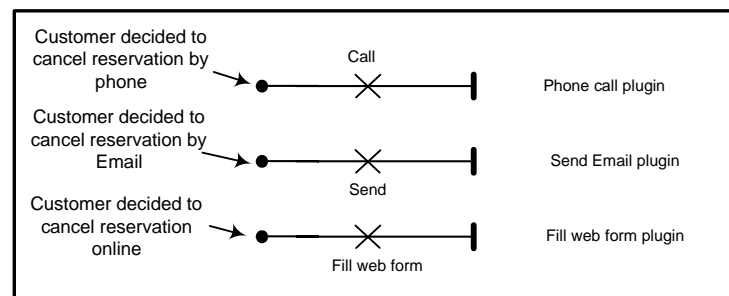


Figure 5.10 Plug-Ins for Cancel Reservation Request Stub

5.3.2 Analysis Phase

The objective of the analysis phase is to transform the system requirements into a representation of the system that can be forwarded to the design phase. The analysis phase is considered to be the most important process of the methodology. This phase starts with analyzing the system requirements phase; it utilizes the system scenario model that is constructed by UML use-cases and use-case maps. This system scenario model is considered as a base to produce the models of the analysis phase. The analysis phase is concerned with the description of the agent architecture as well as the MAS architecture. It is divided into two stages. The first stage describes the agent architecture. The second stage describes the MAS architecture. Figure 5.1 illustrates both architectures. The next sections provide a detailed description of both architectures. The agent architecture stage describes the internal structure (roles,

beliefs, goals, plans and triggers) of agents in the system. In contrast, the MAS architecture stage describes the relationships between agents, the conversations and exchanged messages and agent services. This description of the MAS architecture is important in order to facilitate two main functions:

1. To enable negotiation and cooperation between agents.
2. To establish commitments and agreements that the agents should adhere to in order to provide the services to other agents in the system.

5.3.2.1 Agent Architecture Stage

The agent architecture stage describes the following models:

- Roles model: Discovers the roles that agents play or perform in the system, determines responsibilities for each role and specifies activities for each responsibility.
- Agent model: Identifies agents in the system and assigns roles to them. Refines the roles to fit agent capabilities.
- Beliefs model: identifies agent beliefs.
- Goals model: Identifies agent's goals.
- Plans model: Specifies plans for each goal.
- Triggers model: Identifies the triggers that each agent should be aware of as being events that take place in the system.

The MASD methodology requires the development of all models of the agent architecture stage. They are always developed even if the proposed agent system is just as a single agent.

5.3.2.1.1 Roles Model

The agent role represents an agent behavior that is recognized, providing a means of identifying and placing an agent in a system. Role modeling is appropriate for agent systems [Kendall 1998] because of the following reasons:

- Roles and role models provide a new abstraction that can unify diverse aspects of a system. Software agents, objects, processes, organizations, and people can play roles, and this is especially important in applications that encompass all these types of entities, such as information and process management.
- Role models are patterns that should be documented and shared.
- Role model synergy integrates roles and may be valuable for agent design.
- Role model dynamics can be employed to model mobility, adaptive behavior, context switching, and other aspects of agent systems.

Furthermore, the roles model presents the agent system as an organization by considering it as a set of roles that work together. Each role has its own responsibilities. These roles improve and systematize the agent functionality and emphasize social or interactive behavior. The agent can perform more than one role in the system and more than one agent can perform the role. The roles as encapsulated units can be transferred easily from one agent to another when there is a need.

The roles model is the first task in the analysis phase. In this model, the roles that an agent plays in the system are discovered. It includes the following three detailed

steps: discovering roles, determining role responsibilities, and specifying activities for each responsibility.

5.3.2.1.1.1 Discovering Roles

This step is responsible for identifying the main roles that are found in the system. In order to be able to capture those roles, UCMs and UML use cases scenarios are to be exploited. In the system scenario model, the UCM components that are involved in the system are identified. These components could be agents, objects, or actors. Roles are discovered by analyzing path segments that cross UCM components in the system scenario model. Figure 5.11 illustrates how the roles are extracted from UCMs and UCDs, which have been constructed during the system requirements phase.

This process is performed by passing through all responsibilities and all stubs in all UCM scenarios for each component in the system separately. Roles are also discovered by tracing use cases in UCDs. It is possible then to define the responsibilities and tasks that identify the role or roles which are played by every component of the system (at this moment, only roles are considered and agents are identified later on).

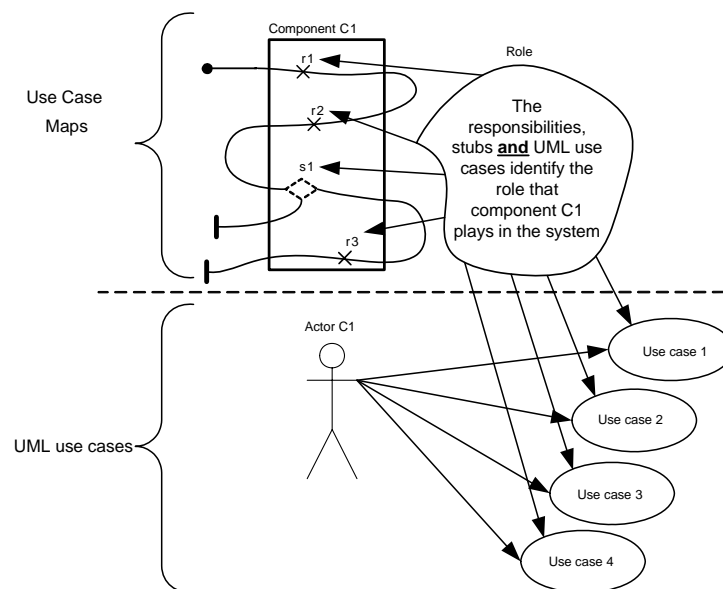
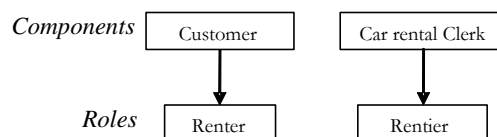


Figure 5.11 Extracting Roles from UCMs and UML Use Cases

Figure 5.12 shows some examples to illustrate how the roles are assigned to UCM components. Components are listed in one row and the roles are listed in a second row. Each role can be associated with one or more components. Each component can be associated with one or more roles.



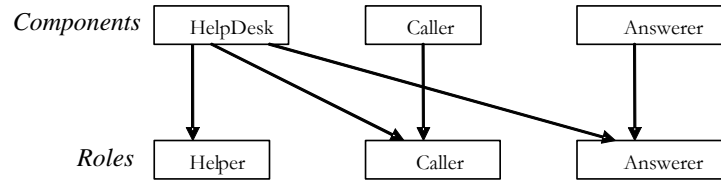


Figure 5.12 Examples of Component-Role Relations

5.3.2.1.1.2 Determining Responsibilities of the Roles

Once roles have been identified then the next step is to determine the duties and **responsibilities** of each role separately. This process starts by tracing scenarios of use case diagrams that have been developed during the system requirements phase, Identifying each actor individually and determine all its use-cases, then transferring them directly (one-to-one) to responsibilities in the role that it plays in the system. The scenario paths of the UCMs are then traversed and all the responsibilities and stubs are individually defined and transferred directly to responsibilities and functions that are carried out by the role. This process is an attempt that most of responsibilities and functions of each role are fully, clearly and accurately captured.

5.3.2.1.1.3 Specifying Activities of Each Responsibility

Once the responsibilities and functions of each role are individually identified, then the following step will identify all the **activities** undertaken by each responsibility. This will in fact, represent the functions of the proposed role to be implemented in the system.

The important attributes of the roles model are: **role name**, **role description**, **responsibilities**, **permissions**, **perceptions**, **obligations** and **constraints**. The **role name** states the name of the role. The **role description** is a textual explanation of the function of the role. **Responsibilities** are the activities that the role is responsible to perform. **Obligations** are requirements that should be available to enable the role to start its functionality and carry out its responsibilities and activities. **Permissions** are the authorities related to numbers and types of resources that will be exploited by agents in the system. **Constraints** are restrictions and boundaries that the role must not violate through executing its tasks. Table 5.1 shows the attributes of the role model. Obligations, permissions, and constraints can be captured from UCM scenarios by the system developers.

Role name:	<i>Role 1</i>
Role description:	<i>Textual description</i>
Responsibilities & its Activities:	<i>Responsibility 1</i> <i>Activity 1.</i> <i>Activity 2. ...</i>
Obligations:	<i>Obl1, Obl2, ... , Obl n</i>
Permissions:	<i>Perm1, Perm2, ... , Perm n</i>
Constrains:	<i>Const1, Const2, ... , Const n</i>

Table 5.1 Role Attributes

Developers systematically apply phrase heuristics to classify the statements as permissions, obligations, or constraints. Heuristics include modality (can, may, must), condition key words (if, unless, except) and English conjunctions (and, or, not). Developers must document their interpretation (e.g., “may” indicates a permission)

and assign logical meanings to each conjunction. Due to logical disjunctions, each sentence may have multiple obligations, permissions, and constraints.

Role name:	Renter
Role description:	Renter who pays rent to use a car that is owned by the car rental company.
Responsibilities & its Activities:	Res.1 Request reservation Act.1 Reserve car by a phone call Act.2 Reserve car by an E-mail Act.3 Reserve car by the Internet Res.2 Cancel reservation request Act.1 Cancel reservation by a phone call Act.2 Cancel reservation by an Email Act.3 Cancel reservation by the Internet Res.3 Notify real customer. Act.1 Notify customer for canceled reservations Act.2 Notify customer for rejected reservations Act.3 Notify customer for confirmed reservations
Obligations:	The renter should pass rental regulations
Permissions:	Null
Constraints:	The renter should not have more than one reservation at the same time

Table 5.2 Renter role for customer component

Once all responsibilities and stubs (request reservation, cancel reservation request, etc.) that the component customer performs have been recognized, then it is quite possible to define and specify the role played by the customer component. Here, it is obvious that it plays a renter role. Table 5.2 illustrates the renter role that the customer component will play in the reservation scenario. In the same situation, the car rental clerk component plays the rentier role. For simplicity, we will describe the renter role in the reservation scenario only. Chapter 6 describes in detail all the roles in the system.

5.3.2.1.2 Agent Model

The agent model describes the internal structure of agents within the system and how these agents employ its internal structure to perform its tasks. In the MASD methodology, the building process of the agent model is based on BDI agent architectures [Georgeff, Pell, Pollack, Tambe, and Wooldridge 1998]. The BDI architecture is used to determine the actions that an agent performs. Each agent possesses one goal or more, which it desires to realize. In addition, an agent has beliefs that it depends on to achieve its goals. It is assumed that agents have a library of goals available to them, each goal containing a set of predefined plans. Each plan contains a set of predefined tasks. Tasks are not necessarily atomic; they could be a single task or a sequence of tasks that form a plan. The term plan is used to achieve a specified goal. Each plan has a set of preconditions and postconditions associated with it. In order for the tasks of that plan to be executable, the preconditions for that plan must be satisfied. These preconditions and postconditions could be considered as the agent's beliefs that it needs to hold in order for it to be able to select the appropriate plan to achieve the goal. Once the plan has been executed, its postconditions are applied. Executing a plan can cause changes to the state of the environment. Agents also have triggers. These triggers assist them to determine the appropriate goal or plan to be selected. The behavior of the agent is determined solely

by its concrete beliefs, goals, and plans. The agent model describes in detail the following steps: Identifying agents, refining roles, beliefs model, goals model, plans model, and triggers model.

5.3.2.1.2.1 Identifying Agents

In this section, the agent identification step is performed to extract those agents that are assumed to exist within the system. These agents are identified using use case maps that have been developed during the system requirements phase. Agents are identified by analyzing UCM components. A component can be identified as an agent or several components that can be combined to constitute one agent. Hence, several roles are combined into one agent.

Fig. 5.13 shows how the roles are assigned to agents. Each agent should be able to fully and logically carry out the specified role or roles assigned to it. Otherwise, the developer must select the most appropriate agent for the role.

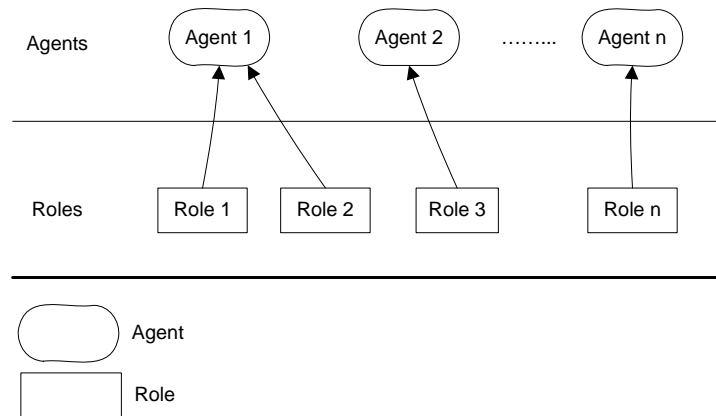


Figure 5.13 Assigning Roles to Agents

In the car rental system the customer and car rental clerk components are assigned respectively to a customer agent and a car rental clerk agent. The car rental manager agent represents the branch manager of the car rental company. This agent can play two roles in the system. The first and main role is a director role. The second role is rentier role. It can play the role of rentier when there is a need for that e.g. when many customers crowd the car rental clerk agent at the same time. Fig. 5.14 shows how more than one the role is assigned to one agent.

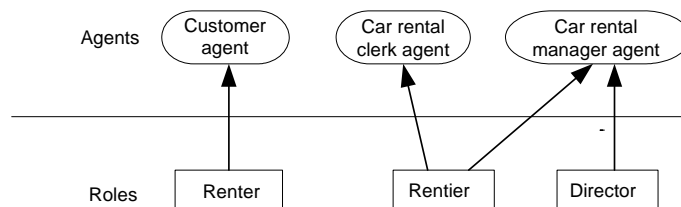


Figure 5.14 Assigning Renter role to Customer agent, Rentier role to Car Rental Clerk and Manager Agents and Director Role to Car rental Manager Agent

5.3.2.1.2.2 Refining Roles

The refining roles step is merely used to revise the roles that the agent plays within the system. The refinement process consists of two steps. The first step is to match the roles that are captured in the roles model with agents that play these roles according to the agent's capabilities. The role's responsibilities are classified based on who is responsible for performing them. The second step is to separate, or isolate those responsibilities that are to be carried out by real persons from those responsibilities that are to be carried out by agents on their behalf.

The refining roles process keeps the responsibilities that are to be carried out by the agent within the roles model. The responsibilities that are to be carried out by real users are stated as preconditions. These preconditions are translated into beliefs. Agents use these beliefs to keep track of whether the real person performs those responsibilities or not. Agents should be able to sense the environment to check whether these beliefs are changed or not. In other words, an agent may wait for a signal (e.g. a message) that confirms that a task performed by the real user has been completed. This refinement process assists developers to build a clear design that is free from confusion and a responsibility overlap.

5.3.2.1.2.3 Agent Beliefs Model

The agent knowledge is considered one of the most important aspects of the agent system. It stores relevant facts about the agent and its environment. Agent knowledge may be taken to explicitly represent the agent's beliefs about its environment or even about itself or about other agents. The following sub-sections show how the agent beliefs are identified. The beliefs model in the MASD methodology is carried out via the system scenario model and the roles model.

The agent beliefs are identified either by the preconditions or by postconditions of the agent's plans and goals, or by the obligations, permissions, and constraints that were obtained in the roles model. Furthermore, the beliefs can be obtained by tracing the UCM scenarios. The stubs and responsibilities are considered as bases of beliefs that are used to trace whether these stubs and responsibilities are achieved by the agent or not. In addition, the beliefs store information about the internal state of agents. Agent beliefs are classified into two types: *constant belief*, these beliefs are set beliefs and not allowed to change, and *variable beliefs*, the values of these beliefs can change many times. Beliefs can be assigned initial values or their values are computed using some kind of expressions or deduced by inference rules. According to Parsons [1998] it is reasonable to assume that the values of the beliefs are obtained in several ways:

- 1) Initial beliefs (basic facts which represent the agent's initial beliefs).
- 2) Beliefs deduced from previous beliefs by deductive inference rules.
- 3) Beliefs obtained as answers to questions put to the environment by the agent.
- 4) Beliefs perceived by a sensor (facts that the agent perceives in its environment).
- 5) Beliefs communicated by external agents (messages received from other agents).

Also MASD classified the purposes of the beliefs as the following: Storage belief, when the belief is stored and the agent can use it during its lifecycle; maintain belief, when the agent must keep the belief at a certain value e.g. when the agent must keep the temperature constantly at 20 degrees; and achieve belief, the agent stores a

required value of the belief and during its lifecycle tries more than one time to check the value of the belief and run plans if the value is not the required value. The agent may not be able to change an achieve belief to a required value but it must keep the value of a maintained belief true at all times. These classifications and its purposes assist the developers to identify the mechanism of how the beliefs are stored and exploited. Accordingly, the agents will be able to reason about the beliefs to select the appropriate actions.

Table 5.3 illustrates the structure of the agent beliefs model. A notification must be made about the beliefs that are captured from obligations, permissions, and constraints of the role. These beliefs are considered as initial beliefs. Therefore, they do not belong to any goals or plans.

Belief	Type	Purpose
Agent Id	Constant	Storage
Bel1	Variable	Maintain
Bel2	Variable	Achieve

Table 5.3 General Structure of the agent beliefs model

The agent beliefs model deals with only some types of beliefs that were mentioned previously. The focus is on those perceived by sensors, those placed as initial beliefs, and those obtained as an answer to questions put to the environment by the agent. The beliefs that are communicated by other agents as the messages received from other agents, are treated as communication messages covered in the agent interaction model. Due to the fact that agents within the system could possess many beliefs, we will provide the beliefs model for the customer agent only for simplicity. Table 5.4 describes the beliefs model for the customer agent.

Belief	Type	Purpose
Agent-Id	Constant	Storage
Customer wants to rent a car	Variable	Storage
Customer decides to reserve by phone	Variable	Storage
Customer decides to reserve by E-mail	Variable	Storage
Customer decides to reserve car online	Variable	Storage
Reservation confirmed	Variable	Storage
Reservation rejected	Variable	Storage
Customer wants to cancel a reservation	Variable	Storage
Customer decides cancel a reservation by phone	Variable	Storage
Customer decides cancel a reservation by E-mail	Variable	Storage
Customer decides cancel a reservation online	Variable	Storage
Cancellation confirmed	Variable	Storage
Reservation already done and car allocated to the customer	Variable	Storage
Cancel reservation is already requested by customer	Variable	Storage
The renter should fit to rental regulations	Variable	Storage
The renter should not have more than one reservation at the same time	Variable	Maintain

Table 5.4 Beliefs Model for Customer Agent

5.3.2.1.2.4 Agent Goals Model

The goal represents a specific target state that the agent is trying to achieve. In a goal-oriented design, goals are considered explicitly as states to be achieved. Therefore, goals also define reasons to execute agent actions. When actions fail, it can be checked if the target state is already achieved, if not, it would be useful to retry the failed action or try out another set of actions to achieve the target state.

The agents' internal structure in the MASD methodology is based on the BDI architecture [Bratman 1987-1999; Rao and Georgeff 1991; Cohen and Levesque 1990; Kinny, Georgeff, and Rao 1996]. The MASD methodology assumes that the concept of goals in relation to agents has a very strong relation with the BDI architecture. The goals represent the desires and intentions that the agent possesses. The definition of the relationship that links goals with the desires and intentions is formulated as being a similarity or matching relationship. Intentions are considered and are defined as being goals that possess previously prepared plans to be executed. Desires are defined as goals with no plans for future execution.

In this model, goals are identified for each agent in the system. These goals represent a mechanism, which leads the agent to perform its actions in an orderly and smooth way. The MASD methodology supports two types of goals (long-term and short-term) in the form of goals and sub-goals. The MASD methodology deals with short-term goals as the goals of the agent, which will be achieved during system runtime. This type of goal is obtained through the methodology process by capturing the agent goals from roles model. More details are available in the agent goals model in the analysis phase. The MASD methodology deals with long-term goals as the strategic goals of the system. This kind of goal cannot be obtained through the methodology process like the short-term goals. They should, however, be deduced by the designer in order to identify the sub-goals (short-range goals) and then determine the conditions of use.

The goals model specifies how to obtain the goals of the agent through the role or roles that it will play within the system. In order to identify the goals of the agent, we have to convert each responsibility of a given role to a specific goal. Therefore, it can be stated that each responsibility within a specific role is considered a goal for the agent who plays the role. Moreover, each activity within a specific responsibility is the foundation for one plan of the goal. Figure 5.15 shows the mapping relationship between the roles and the goals of agents. The transformation relationship from the role responsibilities to the goals is a direct one-to-one relation. Through this step, it is possible to obtain the goals for each agent within the system.

In the agent goals model, the goals that the agent desires to achieve are identified. Each goal and its priorities will be identified. Each goal will be initiated according to its preconditions and a specific priority. The plans, which are prepared by the agent to satisfy the desired goal, will also be identified. This model also contains preconditions and post conditions to initiate the process of achieving goals that the agent desires to realize.

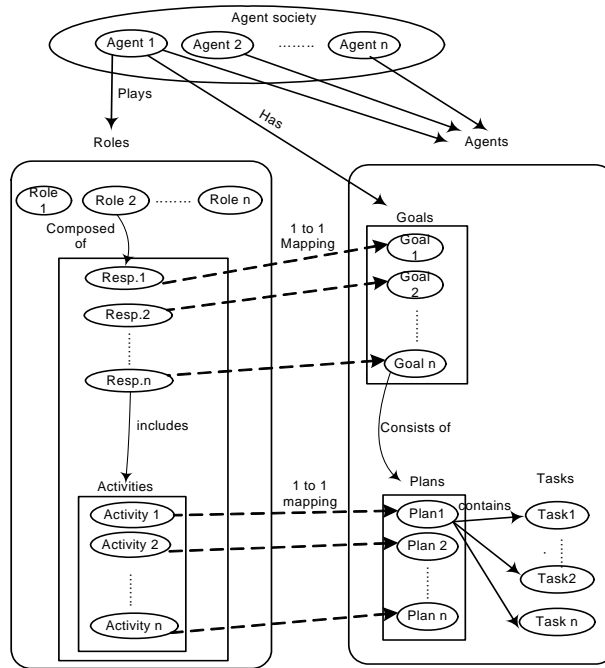


Figure 5.15 Mapping of Roles to Agent's Goals and Plans

Every goal of an agent is composed of a set of attributes that make up its structure. The first field is the **goal** title or the goal name. The second field is the **priority**, which specifies the goal precedence from the execution point of view in cases where there is a need to execute the agent's goals based upon a priority. The priorities are classified as follows: *High*, *above normal*, *normal*, *below normal* and *low*. The third field are **preconditions**. The preconditions are the conditions that must be satisfied in order to consider this goal. The fourth field are the **postconditions**. Postconditions are the conditions that are to be satisfied when the goal is fully achieved. They are considered as an indication showing that the goal has been fully accomplished. Finally, the fifth field are the **plans** through which goals can be achieved. Table 5.5 shows the general structure of the agent goals model.

Goal	Priority	Preconditions	Postconditions	Plans
Goal 1	High	Precondition 1 Precondition 2 Precondition n	Postcondition 1, Postcondition 2, Postcondition n.	<ul style="list-style-type: none"> • <u>Plan 1</u> • <u>Plan 2</u> • <u>....</u> • <u>Plan n</u>
Goal 2	Normal	Precondition 1 Precondition 2 Precondition n	Postcondition 1 Postcondition 2 Postcondition n	<ul style="list-style-type: none"> • <u>Plan 1</u> • <u>Plan 2</u> • <u>....</u> • <u>Plan n</u>

Table 5.5 General Structure of the Agent Goals Model

In the following example, the goals of the customer agent are identified. Table 5.6 illustrates the goals model of the customer agent. That is followed by defining the plans by which each of the goals will be achieved:

Goal	Priority	Preconditions	Postconditions	Plans
Request reservation	High	<ul style="list-style-type: none"> • Customer wants to rent a car 	<ul style="list-style-type: none"> • Reservation confirmed. • Reservation rejected. 	<ul style="list-style-type: none"> • <u>Reserve car by phone call</u> • <u>Reserve car by E-mail</u> • <u>Reserve car online</u>
Cancel reservation request	Normal	<ul style="list-style-type: none"> • Customer wants to cancel reservation 	<ul style="list-style-type: none"> • Cancellation confirmed 	<ul style="list-style-type: none"> • <u>Cancel reservation by phone call.</u> • <u>Cancel reservation by Email.</u> • <u>Cancel reservation online</u>
Notify real customer	High	<ul style="list-style-type: none"> • Real customer must be notified 	<ul style="list-style-type: none"> • Real customer notified 	<ul style="list-style-type: none"> • <u>Notify customer for canceled reservations</u> • <u>Notify customer for rejected reservations</u> • <u>Notify customer for confirmed reservations</u>

Table 5.6 Goals Model for Customer Agent

5.3.2.1.2.5 Agent Plans Model

After the goals of the agents we are identified by the previous step, it is time to describe the plans that should be followed by an agent in order to achieve its goal. Since each agent has a goal or set of goals that it wants or wishes to achieve, a plan or a set of plans for each individual goal must exist. Such a plan needs to be adhered to and followed in order for it to be achieved or performed. Each of those plans consists of a set of tasks to be executed.

5.3.2.1.2.5.1 Specifying Plans for each Goal

Plans are a deliberately prepared means through which agents achieve their goals. A plan is not just a sequence of basic actions, but it may also include sub-goals. Other plans are executed to achieve the sub-goals of a plan thereby forming a hierarchy of plans. The agent keeps track of the actions and sub-goals carried out by a plan to determine and handle plan failures.

Plans are specified by matching and transforming the activities that belong to the responsibilities within the roles. Each plan consists of a set of tasks. These tasks implement the plan and they will complete the required work. Completion and implementation of these tasks is considered the as success of the plan. A given goal is considered to be accomplished if at least one plan related to it was implemented. Plans may be executed in a sequential manner, according to the priority of each plan, or in parallel manner.

In the plans model the plans that should be followed or that have to be selected by an agent during achieving a specific goal are recognized. In other words, every goal has to be achieved through one specific plan or more. Plans are adopted by agents and, once adopted, constrain an agent's behavior and act as intentions. The plans model consists of six parts: a plan name, preconditions, postconditions, successful internal actions, failed internal actions and a plan body. Optional **preconditions**

define the preconditions of the plan, i.e., what must be believed by the agent for a plan to be executable. **Postconditions** are conditions that must be true for the plan when it completes. **Successful internal actions** are the actions that are performed if the plan succeeds. **Failed internal actions** are the actions that are performed if the plan fails. Finally, the **plan body** defines a tree representing a kind of flow-graph of actions to perform. UML activity diagrams [UML Specification 1997] are used to represent the plan body. Activity diagrams are used to model the workflow of the process of the internal operation of the agent system. Activity diagrams illustrate the dynamic nature of the agent system by modeling the flow of control from one activity to another. More details about activity diagrams are described in appendix C.

Executing a plan successfully involves traversing the activity diagram from the start node to the end node. Activity diagrams show the dynamic nature of the system, which is emphasized by the representation of the flow of control between the tasks in the plan. No doubt, that being able to represent these tasks in clear and detailed manner will help developers and programmers in representing and implementing them easily and with more flexibility. Table 5.7 shows the general structure of the agent plans model.

Plan-name:	<i>Plan name</i>
Preconditions:	<i>Cond1, Cond2, ...CondN</i>
Postconditions:	<i>Cond1, Cond2, ...CondN</i>
Successful internal actions	<i>Action1, Action 2, ... Action N</i>
Failed internal actions	<i>Action 1, Action 2, Action N</i>
Plan body	<p>The diagram is a UML Activity Diagram. It starts with a 'Start state' (a solid black circle). An arrow labeled 'Activity' points to 'Task 1' (an oval). From Task 1, an arrow leads to a 'Decision (branch)' (a diamond). From the diamond, a 'Yes' path leads to 'Task 2' (an oval) and a 'No' path leads to 'Task 3' (an oval). Task 2 and Task 3 are connected to a thick horizontal bar representing a 'Concurrent flows' node. From this bar, an arrow leads to another thick horizontal bar, which then leads to 'Task 3'. From Task 3, an arrow leads to an 'End state' (a bullseye circle). Labels 'Alternative flows' and 'Concurrent flows' point to the respective flow types in the diagram.</p>

Table 5.7 General Structure of the Agent Plans Model

The plans for the request reservation goal are constructed. Activity diagrams are used to represent such plans. The following example explains only the *reserve car online* plan of the reservation request goal. All plans for the reservation request goal are described in detail in chapter 6. Table 5.8 illustrates *reserve car online* plan for the customer agent and the tasks that should be performed in this plan. Each activity of the activity diagram represents a task in the plan.

Plan-name:	<i>Reserve cars online</i>
Preconditions:	<i>Customer decides to reserve a car online</i>
Postconditions:	<i>Reservation confirmed</i>
Successful internal actions	<i>Inform the real customer to pick up the car</i>
Failed internal actions	<i>Try with another car rental company</i>
Plan body	<pre> graph TD Start(()) --> A1([Get car Rental website]) A1 --> A2([Read car Rental rules]) A2 --> A3([Verify rules]) A3 --> D{ } D -- "[Accepted]" --> A4([Fill in reservation application form]) A4 --> A5([Approve application]) A5 --> A6([Close car Rental website]) A6 --> End(()) D -- "[Not accepted]" --> D </pre>

Table 5.8 Reserve Car Online Plan for the Customer Agent

5.3.2.1.2.6 Agent Triggers Model

This model identifies and captures triggers that occur during system runtime. The idea of the trigger concept is somewhat similar to the ECA rule (event, condition and action) [Dittrich, Gatzju and Geppert 1995]. Triggers are the events and the changes in the beliefs. All events and the change of beliefs that are expected to occur within the system are identified. This model helps designers and developers to identify these events and select the appropriate reaction for such triggers. Triggers can be caused by information coming from the environment, which has an effect on the behavior of agents. According to that information, the agent performs certain actions as a reaction.

Triggers are obtained by capturing and analyzing the beliefs of each agent that could be changed during runtime. Triggers are also obtained by capturing and identifying the expected events that will occur in the system during runtime. The selection of triggers that prompts a goal or a specific plan is then followed by transferring them into triggers that motivate the agent to perform some given reactions.

The triggers model consists of four attributes: The trigger name, trigger type, trigger activator and the actions. Each trigger is identified by a unique **trigger name**. The **trigger type** can be either an event or a change of belief. The **trigger activator** represents the entity that is responsible for causing such trigger. This entity can be an agent, an object, or a particular resource within the system. **Actions** are either goals to be achieved or plans to be executed. Therefore, actions are labeled accordingly. For

each agent, a trigger model is developed which identifies the triggers that are of interest to it. Table 5.9 describes the general structure of the agent triggers model.

Trigger name	Trigger type	Trigger activator	Actions
<i>trigger1</i>	<i>change of belief event</i>	agent1	• action1 (goal plan).
<i>trigger2</i>	<i>change of belief event</i>	agent2	• action2 (goal plan).
<i>trigger3</i>	<i>change of belief event</i>	Real person	• action3 (goal plan).

Table 5.9 General Structure of Agent Triggers Model

The following example describes the triggers model for the customer agent. By looking at the beliefs model of a particular customer agent and the reservation scenario in UCMs that was built during the system requirements phase, it is found that the scenario begins with belief (precondition) such as “*a customer wants to rent a car*”. In fact, this belief is possible to be true or false. If this belief becomes true, this means the real customer wants to rent a car and this means that the beliefs of the agent have changed. This consequently means that the agent will react based upon the change in its beliefs. This is considered a trigger that motivates the agent to perform a certain action (to start to execute a specific plan or to start to achieve a specific goal such as “*request reservation*” goal) as a reaction. The following table shows a list of the triggers that might occur in the reservation scenario during system runtime. Table 5.10 illustrates the customer agent triggers model.

Trigger name	Trigger type	Trigger activator	Actions
Customer wants to rent a car	Change of belief	Real customer	• Request reservation (Goal).
Customer decides to reserve by a phone	Change of belief	Real customer	• Reserve by a phone (plan).
Customer decides to reserve by an E-mail	Change of belief	Real customer	• Reserve by an E-mail (plan).
Customer decides to reserve online	Change of belief	Real customer	• Reserve Online (plan).
Reservation confirmed	Event	Car rental clerk agent	• Notify real customer to pickup the car (plan).
Reservation rejected	Event	Car rental clerk agent	• Notify real customer about a rejected reservation (plan).
Reservation canceled	Event	Car rental clerk agent	• Notify real customer about a canceled reservation (plan).
Customer wants to cancel a reservation	Change of belief	Real customer	• Cancel a reservation request (Goal).
Customer wants to cancel a reservation by a phone	Change of belief	Real customer	• Cancel a reservation by phone (plan).
Customer wants to cancel a reservation by an E-mail	Change of belief	Real customer	• Cancel a reservation by an e-mail (plan).
Customer wants to cancel a reservation online	Change of belief	Real customer	• Cancel a reservation Online (plan).
Cancellation confirmed	Change of belief	Car rental clerk agent	• Inform a real customer (plan).

Table 5.10 Customer Agent Triggers Model

5.3.2.2 MAS Architecture Stage

The MAS architecture stage is concerned with constructing the multi-agent system. It starts with building the interaction model, which is concerned with capturing all the interaction between the agents in the system. This is followed by constructing the agent relationships model, which is concerned with capturing the relationships between agents in the system. Finally, the agent services model is constructed. This model is concerned with exhibiting the services that each agent should propose to other agents in the system. It facilitates the access to services that are offered by each agent. In addition, it organizes the cooperation between agents in the system. The MAS architecture stage should present the following:

- 1) Identify the interactions between agents in the system by using UCMs scenarios. These interactions explain the process in which agents exchange information with each other (as well as with their environment).
- 2) Capture the relationships between agents in the system in order to assist agents to identify dependencies between them.
- 3) Capture the services that each agent should provide in the system.

5.3.2.2.1 Agent Interaction Model

The agent interaction model specifies all interactions between agents in the system. The agent interaction model explains the process in which agents exchange information with each other and with their environment. It describes the agents' conversations as a set of high-level interactions and as an initial step for communicating between agents. A more detailed description of these interactions through *interaction protocols* will be fully specified at the design phase. To construct such communication among agents, a common language has to be developed. This model is considered as an initial model in the analysis phase to represent interactions between agents in a high-level view. In the design phase, the FIPA-ACL is used to represent interactions in more detail.

The agent interaction model is represented by a notation called interaction diagrams. This notation is suggested by the MASD methodology to describe such interactions. The main function of interaction diagrams is to transform the use case maps scenarios that are developed in the system scenario model into communication messages between the agents. These communications should be comprehensible by the system's agents. Therefore, a simple agent conversation language is developed based on the speech act theory by Tsohatzidis [1994] to help agents understand each other. This speech act theory consists of a communicative act called *performative* which means purposeful actions performed during conversations between the communicators. For example, the *request performative* means that the sender requests the receiver to execute some action/actions. On the other hand, the receiver can recognize which type of response is expected from the contents of the conversation. Table 5.11 illustrates each performative and its description that are used in the agent conversation language.

Performative	Description
Request	The sender requests the receiver to execute some actions.
Query	The sender asks the receiver about some information the sender does not know.
Inform	The sender gives the receiver some information the sender knows.
Provide	The sender provides the receiver with information already requested by the receiver.
Call for proposal (cfp)	The sender calls for proposals of executing some actions.
Propose	The sender proposes to execute specific actions under some preconditions.
accept-proposal	The sender accepts the proposal to execute some actions presented in advance.
reject-proposal	The sender rejects the proposal presented in advance.
Agree	The sender agrees to execute some actions.
Reject	The sender rejects to execute some actions because the sender cannot execute them.
Failure	The sender notifies that it tried to execute some actions but the execution has failed for some reason.
not-understood	The sender notifies the receiver that it cannot understand the message the sender received.

Table 5.11 Performatives for Agent Conversation Language

Interaction diagrams are developed from the system scenario model by capturing the lines that connect agents (components) in the use case maps diagram and transfer them into conversations (communication acts) between the agents.

The interaction diagram consists of the following notations: The black circle refers to the starting point of the interaction. The path indicates the flow of events in the interaction. The arrow indicates the direction of the flow. The title indicates the performative or the event being exchanged. The symbol “X” indicates the end of the interaction. The agent life bar indicates the life of the agent in the interaction. Figure 5.16 shows the notation of interaction diagrams.

Figure 5.17 illustrates the mapping from UCMS scenarios to interaction diagrams and shows how interaction diagrams are derived from UCMs scenarios. It describes the request of the customer agent to the car rental clerk agent, which then requests more detailed information about the customer. The customer agent replies to the car rental clerk agent who checks the rules of the car rental company and then replies either by acceptance or by rejection

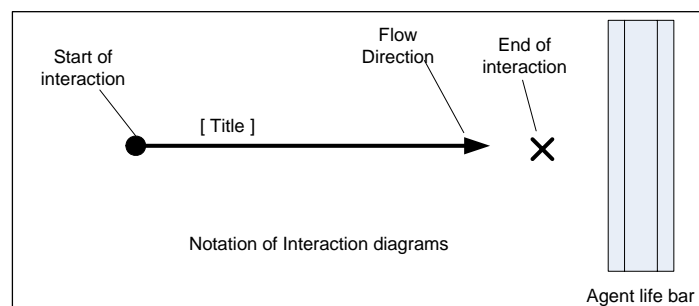


Figure 5.16 Notation of Interaction Diagrams

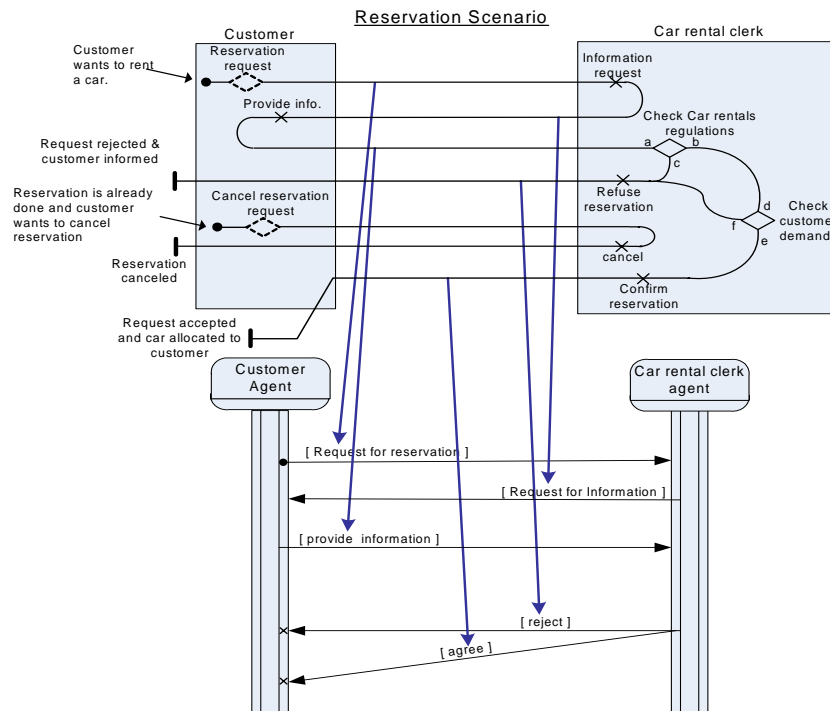


Figure 5.17 Mapping from UCMS Scenarios to Interaction Diagrams

Agent interaction diagrams give only a partial picture of the system behavior. In order to have a precisely defined system it is necessary to progress from interaction diagrams to interaction protocols. Interaction protocols define precisely which interaction sequences are valid within the system.

5.3.2.2.2 Agent Relationship Model

The agent relationships model describes relationships between the agents. It helps the agents to make the necessary decisions when cooperation between agents takes place. It also establishes an official framework of duties and responsibilities. The agent relationships model consists of a set of system components (agents, objects, resources, etc.) that are connected together to satisfy and pursue a common goal. In this model, the dependencies, and authorities between the system components are described as well as the constraints and restrictions that the system must not violate. The model assists in organizing the coordination between the system agents. This coordination is achieved through a set of commitments realized by formal agreements and contracts that guarantee rights for both parties. The complete set of these commitments comprises a contract. Each commitment is directed from one agent giving this commitment towards its contracting partner, who receives this commitment. In addition, this model helps to identify the proper communication protocols that will be chosen for the conversation between agents in the design phase.

The concept of dependency relationships was inspired from Elammari et al. [1999], and Yu [1995; 1994] for capturing several types of constraints and relationships that are frequently encountered in business processes. Agents' dependency relationships are represented as diagram, where each square represents an

agent, and each link between two agents represents the relationship. The link between two agents indicates that one agent (dependant) depends on the other (dependee) to do something in order that the dependant may achieve some goal. The depending agent is called the dependant, and the agent who is depended upon is called the dependee. The dependency relationship object is called the dependum. Examples of dependencies are goals to be achieved and tasks to be performed. Three types of agent dependency relationships are identified: goal, task, and resource dependency. These relationships can be established either by runtime negotiation or advanced commitment. The model distinguishes among the types of restrictions based on the type of the required relationship between dependant and dependee dependencies. Fig. 5.18 illustrates the symbols that are used for agent dependency relationships. An arrow represents dependency that is going from a dependee agent to a dependent agent.

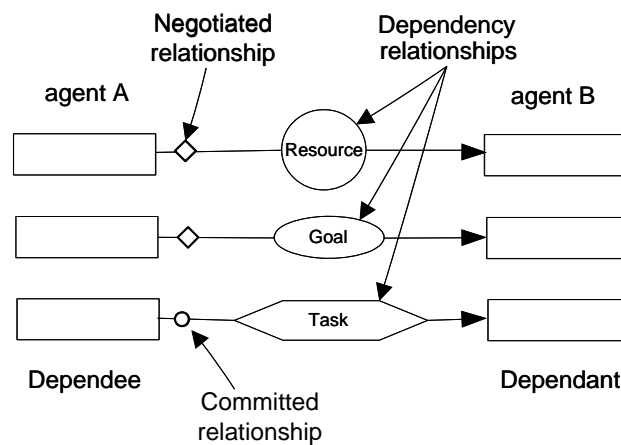


Figure 5.18 Dependency Relationship Symbols

Task dependency represents a relationship in which an agent requires a specific task to be performed. **Goal dependency** represents the relationship in which an agent is dependent on another agent to achieve a specific goal. **Resource dependency** represents the relationship in which an agent is dependent on a supplying agent to provide it with a specific resource. A resource can be physical or informational. These three types of dependencies can be either a **negotiated** or a **committed** relationship. **Negotiated** relationships represent a relationship where an inter-agent negotiation is required to fulfill the dependency. **Committed** relationships indicate that an agent is obligated to provide a service to fulfill the dependency.

Fig. 5.19 shows dependencies between the customer agent and the car rental clerk agent. The customer agent dependencies are stated first, and then the reservation agent dependencies. The customer agent depends on the car rental clerk agent to handle reservation requests. This dependency is classified as “goal dependency” because the customer agent depends on the reservation agent to achieve a specific goal. This goal is called “request reservation”. It also depends on the car rental clerk agent to achieve the canceling reservation goal when the customer wishes to cancel the reservation, or to provide him/her with his/her list reservation information. The car rental clerk agent depends on the customer agent to provide him with the personal information.

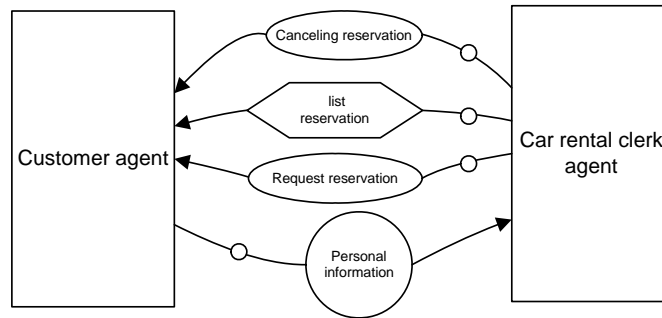


Figure 5.19 Dependency Diagram between Customer Agent and Reservation Agent

5.3.2.2.3 Agent Services Model

The agent services model provides a standard mean of interoperation between agents in the system. This model is intended to provide a common description of agent services. The model is also intended to define the location of the agent services within a multi-agent system. This guides the agent community to find those services easily. A service is provided by an agent and is used by another agent. Agent services are captured by means of the messages exchanged between requester agents and provider agents. The main goal of the agent services model is to facilitate access to services that are offered by each agent. Moreover, it organizes the cooperation between agents through constructing formal agreements. An agreement maintains the agents' rights by providing them the ability to obtain those services in time.

This model is composed of the following five parts: **Service**, **agent**, **expiry date**, **time of availability**, and **cost**. The **service** represents the service title. The **agent** represents the agent offering the service. The **expiry date** represents the end date of the service. The **time of availability** is the time that the service should be available to be exploited by other agents. The **cost** represents the service cost. The agent service model is derived from the use case diagrams that were developed in the system scenario model. Agent services can be derived directly from use case diagrams where each use case can be identified as service. In addition, agent services can be identified as a set of use cases that are compounded into one service. Table 5.12 illustrates the agents' services model including the car rental clerk agent services.

Service	Agent	Expiry date	Time of availability	Cost
Reply to customer inquiries	Car rental clerk agent	Open	always	Free
Handle reservation request	Car rental clerk agent	Open	8:00 am to 8:00 pm	Free
Handle rental	Car rental clerk agent	Open	8:00 am to 8:00 pm	Free
Handle car service	Car rental clerk agent	Open	8:00 am to 4:00 pm	Free

Table 5.12 Agent Services Model

Providing agent services allows agents to search for a certain service that it requires in order to complete its goals or tasks. This model may be updated at runtime by new agents with their services or by new services for agents that already exist.

5.3.3 Design Phase

The design phase introduces the detailed representation of the models developed in the analysis phases and transforms them into design constructs. These design constructs are useful for actually implementing the new multi-agent system. The models that were developed in the analysis phase are revised according to the specification of implementation. The main objective of the design phase is to capture the agent structural design and system design specifications. The design phase has three steps:

1. Creating an agent container.
2. Defining an inter-agent communications.
3. Creating a directory facilitator.

The design phase deals with the concepts that have been developed in the analysis phase and illustrate how these concepts can be designed by identifying how to handle agent's beliefs, goals, and plans, as well as state how to compose the agent *capabilities* into reusable agent modules. In addition, it specifies the inter-communication among agents and how these agents cooperate in order to realize a common goal. A Directory Facilitator (DF) mechanism is also described.

5.3.3.1 Agent Container Model

The first step of the design phase is to construct the agent container, which can be seen as a type specification for a class of instantiated agents. An agent container represents agent behaviour, which can be modularized and decomposed into role specifications that are used by an agent. The core part of the agent specification is to define beliefs, goals, plans and capabilities of the agent and place them in the appropriate agent part.

The agent behavior is defined by a container that represents agent roles and its conversations. The agent container simply contains all the important aspects that are needed by the agent to start working. The agent container is composed of several components (beliefs, goals, plans, and triggers) where each is represented by a certain model. Each model and its programming aspects will be designed in order to fit with the Jadex framework.

5.3.3.1.1 Beliefs

The first part of the agent container are the agent beliefs. The beliefs are considered as “agent beliefbase” which represents the agent knowledge about the environment or the world in which the agent works. The beliefbase is the container of the facts known by the agent. The agent's beliefbase can be considered as simple data-storage, responsible for creating new beliefs, belief sets, or removing old ones. This beliefbase is shared among all agents' plans. The beliefs are classified into two types: Beliefs that allow the storage of exactly one fact and beliefs that allow the storage of a related set of facts.

More details are added to the beliefs during the design stage. A new field called **class** is added to indicate that the type and the possible values are: *Integer*, *string* or *boolean*. The **initial value** for the field depends on the type of belief. The **category** field was also added which refers to “F” for the beliefs that store exactly one fact and refers to “S” for the belief that store set of facts. These additional fields assist the

designers to specify the way in which these beliefs are implemented correctly. Table 5.13 provides a detailed description of all the additions of customer agents done during the analysis stage.

Belief	Type	Purpose	Class	Initial value	Category
Agent_Id	Constant	Storage	String	Customer agent	F
Customer wants to rent a car	Variable	Storage	Boolean	True	F
Customer decides to reserve by phone	Variable	Storage	Boolean	True	F
Customer decides to reserve by E-mail	Variable	Storage	Boolean	True	F
Customer decides to reserve car online	Variable	Storage	Boolean	True	F
Reservation confirmed	Variable	Storage	Boolean	True	F
Reservation rejected	Variable	Storage	Boolean	True	F
Customer wants to cancel reservation	Variable	Storage	Boolean	True	F
Customer decides cancel reservation by phone	Variable	Storage	Boolean	True	F
Customer decides cancel reservation by E-mail	Variable	Storage	Boolean	True	F
Customer decides cancel reservation online	Variable	Storage	Boolean	True	F
Cancellation confirmed	Variable	Storage	Boolean	True	F
Reservation already done and car allocated to the customer	Variable	Storage	Boolean	True	F
Cancel reservation is already requested by customer	Variable	Storage	Boolean	True	F
The renter should fit to rental regulations	Variable	Storage	Boolean	True	F
The renter should not have more than one reservation at the same time	Variable	Maintain	Boolean	True	F

Table 5.13 Revised agent beliefs model

5.3.3.1.2 Goals

The second component in the agent container are agent goals. The goals that we developed during the analysis stage are classified into four types of goals: perform, achieve, query, and maintain goal types. **Perform goal** is a type of goal where some action is required to be performed. The results of the goal depend on specific actions. Naturally, when no actions could be performed, the goal has failed. Otherwise, when one or more plans have been executed the goal is successful. **Achieve goal** is a goal where an agent wants to achieve a certain state (target state) of affairs. This target state is represented by a target condition. When an agent gets a new achieve goal (e.g. no waste at given location) that shall be pursued, the agent starts activities for achieving the target state. When the target state is reached then the goal has been achieved. Otherwise, for a yet unachieved goal, plans are selected for execution. Whenever during the plan execution phase, the target condition is reached, then all running plans of that goal can be aborted. **Query goal** is used to enquire information about a specified issue. Therefore, the goal is used to retrieve a result for a query and does not necessarily cause the agent to engage in actions. When the agent has sufficient knowledge to answer the query the result is obtained instantly and the goal succeeds. Otherwise, applicable plans will try to gather the needed information. **Maintain goal** is the goal that has to keep a specific desired state (its maintain condition) satisfied all the time. When the condition is not satisfied any longer, plans are invoked to re-

establish the given state. The maintain goal stays idle until the maintained condition is violated. An example for a maintain goal is to keep the temperature of a nuclear reactor below some specified limit. When this limit is exceeded, the agent has to act and normalize the state.

Based on this classification, we added a **type** field to the goal model as shown in the following goal model. Table 5.14 provides a detailed description of the added type field of the agent goals model that was obtained in the analysis phase.

Goal	Type	Priority	Preconditions	Postconditions	Plans
Request reservation	Achieve goal	High	<ul style="list-style-type: none"> Customer wants to rent a car 	<ul style="list-style-type: none"> Reservation confirmed Reservation rejected 	<ul style="list-style-type: none"> <u>Reserve car by phone call</u> <u>Reserve car by E-mail</u> <u>Reserve car online</u>
Cancel reservation request	Achieve goal	Normal	<ul style="list-style-type: none"> Customer wants to cancel reservation 	<ul style="list-style-type: none"> cancellation confirmed 	<ul style="list-style-type: none"> <u>Cancel reservation by phone call</u> <u>Cancel reservation by Email</u> <u>Cancel reservation online</u>
Notify real customer	Achieve goal	Normal	<ul style="list-style-type: none"> Real customer must be notified 	<ul style="list-style-type: none"> Real customer must be notified 	<ul style="list-style-type: none"> <u>Notify customer for cancelled reservations</u> <u>Notify customer for rejected reservations</u> <u>Notify customer for confirmed reservations</u>

Table 5.14 Revised agent goals model

5.3.3.1.3 Plans

In this section, we refine the plans that have been developed in the analysis phase in order to meet the design specifications. The plans are classified into two types. The first type is called the **service plan**; a plan that has service nature. An instance of the plan is usually running and waits for service requests. It represents a simple way to react on service requests in a sequential manner without the need to synchronize different plan instances for the same plan. Therefore, a service plan can organize its tasks in a queue for later processing, even when it is busy working. The second type is called the **passive plan**. This type can be found in all other procedural reasoning systems. Usually, the passive plan is only run when it has a task to achieve. For this kind of plan, triggering events and goals should be specified to let the agent know what kinds of events the plan can handle (as represented in the agent triggers model). When an agent receives an event, the candidate plan(s) should be selected and instantiated for execution. We add a field called **type** to the plan, which identifies the type of the plan. This field helps developers to decide the suitable mechanism for plan implementation. Table 5.15 shows the same table that was shown in the analysis phase plus an additional field called type.

Plan-name:	<i>Reserve cars online</i>
Type:	<i>Passive plan</i>
Preconditions:	<i>Customer decides to reserve car online</i>
Postconditions:	<i>Reservation confirmed</i>
Successful internal actions	<i>Inform the real customer to pickup the car</i>
Failed internal actions	<i>Try with another car rental company</i>

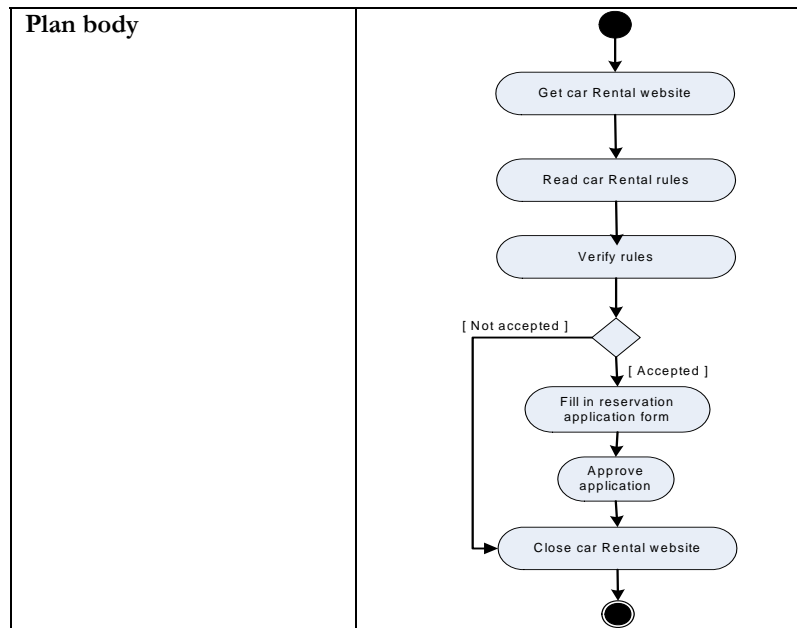


Table 5.15 Reserve car online plan with Plan Type Field

5.3.3.1.4 Capabilities

In many situations, goals, beliefs and plans become a common part of a specific task. The agent may need to achieve more than one goal, use more than one belief, and plan to perform a certain task in the system. Capabilities are simply a group of goals, beliefs and plans grouped together in a package in order to be used when they are needed to do a certain task in the system. These capabilities are captured from the identified beliefs, goals and plans that are required by the agent capability to implement a specific task. The agent capability is derived from the roles model that has been developed in the analysis phase. The following structure shows how the reservation capability is structured.

Reservation capability:

Begin // *Reservation capability,*

Beliefs:

*Customer wants to rent a car,
Reservation confirmed,
Reservation rejected.
Customer wants to cancel reservation
Cancellation confirmed.*

Goals:

Begin // *goals*

Request reservation goal

Plans:

*Reserve by phone call,
Reserve by E-mail,
Reserve car online,*

Cancel reservation request goal.

Plans:

Cancel reservation online.

Cancel reservation by phone call.

Cancel reservation by Email.

End // goals

End // Reservation capability

5.3.3.1.5 Triggers

In the design phase, triggers that were developed in the analysis phase are handled as conditions that trigger plans or goals. For example, the goal to keep a reactor temperature below a certain level is a goal that is triggered whenever the temperature exceeds the normal operating level. At runtime, plans are instantiated to handle events and to achieve goals. Activation triggers of the plan/goal are used to specify if a plan/goal should be instantiated when a certain event occurs. Plans are declared by specifying how to instantiate them from their class. For passive plans to be instantiated on demand, a trigger has to be stated. The trigger can be omitted in the case of a plan to be executed, when the agent starts (initial plan).

5.3.3.2 Inter-Agent Communication Model

This model describes in detail possible interactions between agents. To establish communication between agents, agreed on and accepted protocols have to be deployed. The most established standard is the FIPA Agent communication language [FIPA-ACL]. More details about it are stated in appendix D. It describes the conversations between agents in more detail than the agent interaction model.

This model is derived by transforming the interaction diagrams that were developed in the agent interaction model in the analysis phase into conversation messages according to FIPA protocol patterns. The flow of interaction diagrams is transferred into messages according to the FIPA ACL protocols. The interaction diagrams are classified according to FIPA ACL protocols that the agents should follow to realize successful conversations. Determining the proper protocol for each interaction diagram is considered as the bases in the process of selecting the proper message between agents. Figure 5.20 illustrates how an interaction diagram is transferred into messages according to FIPA ACL protocols.

The exchanged messages between the agents are considered as a FIPA ACL message. A FIPA ACL message is composed of a set of one or more message parameters. The parameters are needed for an effective agent communication and will vary according to the situation. The only parameter that is mandatory in all ACL messages is the performative. However, usually ACL messages will also contain a sender, a receiver and content parameters.

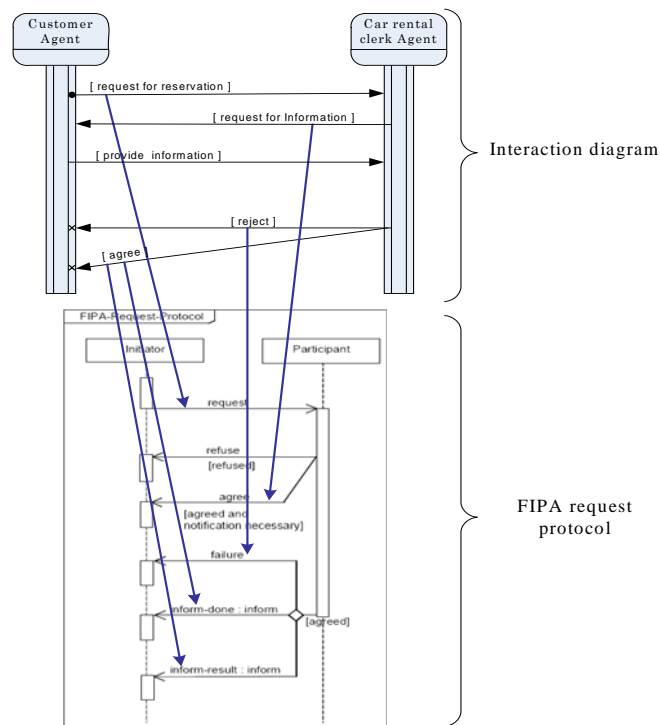


Figure 5.20 The Correspondence between Interaction Diagrams and FIPA Protocols

The contents expression of a message can be handled by the FIPA content language. It includes several sub-languages such as: FIPA-SL Semantic Language which is composed of first order logic and modal operators for mental terms and uncertainty; FIPA-CCL Content Constraint Language which describes the knowledge in terms of a Constraint Satisfaction Problem and FIPA-KIF Knowledge Interchange Format which is essentially first order logic supported by some second order constructs. Using this content language each agent should be able to understand the content of the message and be able to manage its own knowledge (knowledge base) based on a specific ontology (domain specific) to identify the meaning of the message. The following example represents a sample of the messages that are exchanged between the customer agent and the car rental clerk agent with some parameters:

In the following example, the customer agent requests the car rental clerk agent to reserve a car.

```
(request
:sender (agent-identifier :Customer_agent)
:receiver (agent-identifier :Car_rental_clerk_agent)
:content
  "Reserve group B car for rent"
:reply-with reserve-car
:language sl
:ontology e-Rent
:protocol fipa-request interaction)
```

In the following message the car rental clerk agent answers the customer agent that it agrees to the request.

```
(agree
:sender (agent-identifier :Car_rental_clerk_agent)
```

```

:receiver (agent-identifier :Customer_agent)
:content
  ((action (agent-identifier :Car_rental_clerk_agent)
    (agree))
:protocol fipa-agree
:language fipa-sl)

```

5.3.3.3 Directory Facilitator Model

The final step of the design phase is building the Directory Facilitator (DF) model. This model is an extension of the agent services model, which was developed in the analysis phase. The DF model serves as the “yellow pages” to the system agents. The DF allows agents to publish one or more services they provide so that other agents can find and successively use it. Agents may register their services with the DF or query it to find out what services are offered by which agents. An agent is responsible to provide information related to service e.g. service type, service name etc. Furthermore, an agent can also deregister or modify its service details. Any agent can interact with a DF to make its services public and to identify agents that provide a particular service through the yellow pages. In addition, agents can ask (search) the DF looking for agents, which provide the services they desire. The DF should provide the agents in the system with the following functions: **register**, **deregister**, **modify** and **search**.

5.3.3.3.1 Directory Facilitator Mechanism

Every agent that wishes to advertise its services to other agents should find DF and request the registration of its agent description. There is no intended future commitment or obligation on the part of the registering agent implied in the act of registering. For example, an agent can refuse a request for a service, which is advertised through a DF. Additionally, the DF cannot guarantee the validity or accuracy of the information that has been registered with it; neither can it control the life cycle of any agent. The service description must be supplied containing values for all of the mandatory parameters of the description (as we have shown in the service agent model). It may also supply optional and private parameters that an agent developer might want to include in the directory.

An agent may search in order to request information from a DF. The DF does not guarantee the validity of the information provided in response to a search request, since the DF does not place any restrictions on the information that can be registered with it. However, the DF may restrict access to information in its directory and will verify all access permissions for agents, which attempt to inform it of agent state changes.

5.3.4 Implementation Phase

The implementation phase is the point in the development process when we actually start to develop the program code. During the implementation phase, the system is built according to specifications from previous phases. Previous phases provided models that can be transferred into an implementation. The produced models have a set of design specifications showing how the agent system and its components should be structured and organized. The design specifications are used to develop the implementation phase. There are several agent frameworks and platforms

proposed to develop multi-agent systems. MASD supports some of them such as JADE [1999], JACK [Busetta et al. 1999], MADKIT [1999], Jason [Bordini et al. 2005], and Jadex [Braubach et al. 2004] as a tool for the development process. We recommend the Jadex platform because it is Java based, has a FIPA compliant agent environment, and allows developing goal-oriented agents following the BDI model. More details about the Jadex framework are stated in appendix E. We can show how Jadex handles the design models. The customer agent is used as an example to show how agents are implemented in Jadex. This section describes in brief how the customer agent is implemented. More implementation details will be described in chapter 6.

Due to matter of time and scope, we will not discuss how to set up the Jadex environment as it can be done in a few simple steps. Starting up an agent begins with the creation of the agent. The agent is created according to the agent container that was developed in the design phase. Each agent container represents an Agent Definition File (ADF) in Jadex.

Firstly, we create a new agent definition file (ADF) called **customer.agent.xml**. In this file, all important agent startup properties are defined in a way that complies with the Jadex schema specification. The first attribute of the agent is its type name, which must be the same as the file name (similar to Java class files). In this case, it is set to **Customer**. Additionally one can specify a package attribute, which has a similar meaning as in Java programs and serves for grouping purposes only (the package name will need to be altered with respect to the actually used directory structure). All plans and other Java classes from the agent's package are automatically known and need not to be imported via an import tag. The following XML code describes in brief the customer ADF.

```

<!-- CustomerAgent -->

<agent xmlns="http://jadex.sourceforge.net/jadex"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://jadex.sourceforge.net/jadex
      http://jadex.sourceforge.net/jadex-0.96.xsd"
      name="Customer"
      package="jadex.tutorial">
  <beliefs>
    <belief name="agent_Id" class="String">
      <fact>"customer1"</fact>
    </belief>
    <belief name="customer_wants_to_rent_a_car" class="Boolean">
      <fact>"true"</fact>
    </belief>
    <belief name="customer_decide_to_reserve_car_by_phone" class="Boolean">
      <fact>"true"</fact>
    </belief>
    <belief name="customer_decide_to_reserve_car_by_email" class="Boolean">
      <fact>"false"</fact>
    </belief>
    <belief name="customer_decide_to_reserve_car_online" class="Boolean">
      <fact>"false"</fact>
    </belief>
    <belief name="Reservation confirmed" class="Boolean">
      <fact>"false"</fact>
    </belief>
  </beliefs>
  <goals>
    <achievegoal name="request_reservation">
      <creationcondition>
        $beliefbase.Customer_wants_to_rent_a_car.
      </creationcondition>
      <unique/>
      <deliberation>

```

```

        <inhibits ref="cancel_reservation_request "/>
    </deliberation>
</targetcondition>
    $beliefbase.reservation_confirmed || $beliefbase.reservation_rejected
</targetcondition>

</achievegoal>
<achievegoal name="cancel_reservation_request">
    <creationcondition>
        $beliefbase.customer_wants_to_cancel_reservation.
    </creationcondition>
    <unique/>
</achievegoal>

</goals>

<plans>

<plan name="reserve_car_by_phone_call">
<body> new reserve_car_by_phone_callPlan() </body>
<trigger>
    <condition>
        $beliefbase.customer_decide_to_reserve_car_by_phone
    </condition>
</trigger>
<contextcondition>
    $beliefbase.!reservation_confirmed || $beliefbase.!reservation_rejected)
</contextcondition>
</plan>

<plan name="reserve_car_by_email">
<body> new reserve_car_by_emailPlan() </body>
<trigger>
    <condition>
        $beliefbase.customer_decide_to_reserve_car_by_email
    </condition>
</trigger>
<contextcondition>
    $beliefbase.!reservation_confirmed || $beliefbase.!reservation_rejected)
</contextcondition>
</plan>

<plan name="reserve_car_online">
<body> new reserve_car_onlinePlan() </body>
<trigger>
    <condition>
        $beliefbase.customer_decide_to_reserve_car_online
    </condition>
</trigger>
<contextcondition>
    $beliefbase.!reservation_confirmed || $beliefbase.!reservation_rejected)
</contextcondition>
</plan>

</plans>
</agent>

```

5.4 Chapter Summary

In this chapter, the Multi-Agent System Development (MASD) methodology was detailed and the methodology construction necessities were stated. The phases of the proposed methodology were described and the system requirement phase was built. UCMs and UCDs were used as a notation to build the system requirements phase. The system functionality was partitioned into transactions meaningful to users and developers of a system. The analysis phase was constructed and the system requirements were transferred into a representation of the agent system that can be forwarded to the design phase. A set of essential models was developed. Some of these models represent the agent architecture stage such as agent model, roles model, goals model, plans model, beliefs model and triggers model. The other models represent a MAS architecture such as the interaction model, agent relationships model

and agent services model. These models were then used in the design phase. The design phase was a discussion of how to design agents and agent's roles, plans, beliefs, and goals, as well as the agent capabilities. The chapter also described how to specify the inter-communication among agents and described the Directory Facilitator (DF), as well as its mechanism and function. Finally, the implementation phase was discussed. This phase was the development process to construct the solution and writing program code. The design specification was captured to build the implementation step and the Jadex platform was used as a tool to implement the agent system. In the next chapter, we will illustrate our methodology using a complete case study.

CHAPTER SIX

CASE STUDY: CAR RENTAL SYSTEM

6.1 Introduction

The chapter describes the entire detailed process of developing multi-agent systems using a case study of the car rental system. This case study is used to prove the methodology. In this chapter, we provide a detailed description of how the car rental system works which represents the case study to test and evaluate the new methodology.

6.2 Case Study: Car Rental System

The case study “car rental system” has been chosen because it is simple and straightforward. It can be used to illustrate the types of reflective reasoning required by agents involved in a distributed collaborative environment. It entails a distributed design process, where several participants need to interact with each other. It encompasses and highlights a number of underlying and interconnected agent concepts.

In an example scenario, EU-Rent is a car rental company owned by the EU-Corporation [EU-Rent]. It is one of three businesses. The other two businesses are hotels and an airline. Each has its own business and IT system, but with a shared customer base. Many of the car rental customers also fly with EU-Fly and stay at EU-Stay hotels. This case study was developed by Model Systems, Ltd., along with several other organizations, and has been used by other organizations [Hay and Healy 2000].

EU-Rent is a car rental company with branches in several countries. It provides typical car rental services:

- Different types of cars are offered, which are organized into groups. All cars that are in one group are charged at the same rate.
- Cars may be rented through reservations made in advance or by “walk-in” customers on the day of rental.
- Cars are picked up from EU-Rent branches at the start of a rental, and may be returned to the same or a different branch.
- Customers may join the EU-Rent loyalty club, and accumulate points that they can use to pay for rentals.
- EU-Rent from time to time offers discounts and free upgrades subject to conditions.

EU-Rent records “bad experiences” with customers such as speeding fines or damage to the car during rental and may refuse subsequent rental reservations from such customers. In this phase, we deal with a part of EU-Rent’s behaviour, triggered by the following kinds of event:

1. Accepting new reservations for new and existing customers.
2. Assigning cars to the day’s rental agreements.
3. Selection of the best discount that the rental agreement qualifies for.

4. Handling reservations where the requesting customer has had “bad experiences”.
5. Transfer of the ownership of a car when a car is returned to a branch different from the pick-up branch.

The case study is considered to be applied with both UCMs and UCDs.

6.3 System Requirement Phase

In this phase, UML Use-Case Diagrams (UCDs) and Use-Case Maps (UCMs) are used to describe the car rental system requirements in high-level visual representations. The system requirement phase concentrates on constructing the system scenario model for the car rental agent system. Two techniques are specifically used: UML use-case diagrams and use case maps. More details about these techniques are described in appendixes A and B.

6.3.1 System Scenario Model

The main task of the system requirement phase is the construction of the system scenario model. The scenarios of the car rental system are established. This system is considered as a small-distributed system. The system scenario model is composed of two steps: developing UCDs for the car rental system, and developing UCMs for the car rental system.

6.3.1.1 UCDs for Car Rentals System

In this section, a detailed example will be provided which performs the construction of UCDs scenarios of the EU-Rent a car rental system. Each use case in the system scenario will be described with a diagram as well as describing and clarifying its components. Initially, use case diagrams of the dialogues for the car rental system scenario will be created as in figure 6.1. It will be followed by a description of each use case separately.

In the UCDs, we should capture the following system components:

1. The actors involved in the system.
2. The use cases performed by actors.
3. In each use case, we should perform the following tasks:
4. Identifying the description of use case.
5. Identifying pre-conditions and post-conditions of each use case.
6. Identifying the goal of the use case.
7. Identifying the actor that performs the use case.
8. Identifying triggering events of the use case.
9. Identifying extensions and alternatives of the use case.

The car rental system includes three actors: Customer actor, car rental clerk actor and car clerk actor. The duties of these actors are described as use cases. Figure 6.1 illustrates the UCM for the car rental system. The numbers of actors in the car rental system can be more than three. Manager and cashier actors can be added to the system. However, in order to make the case study simple and easy to understand, we selected the most important ones to represent the system.

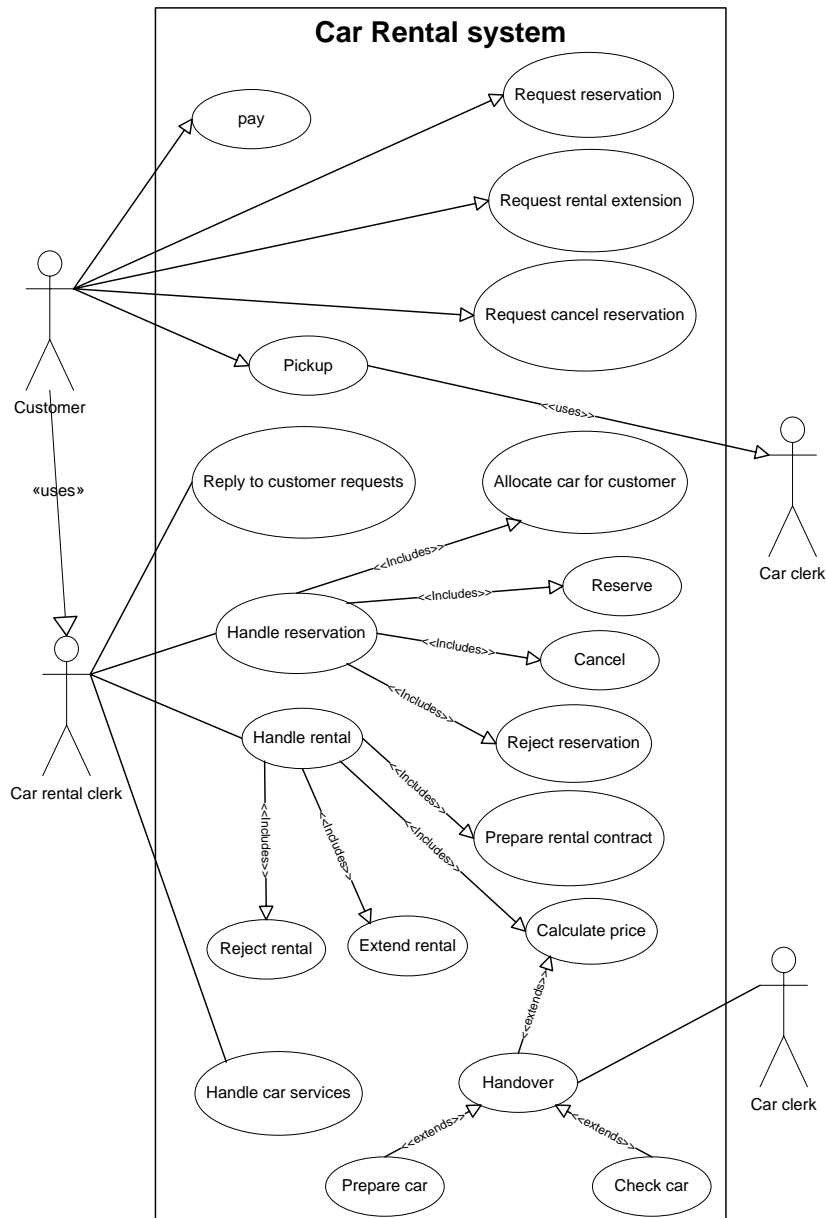


Figure 6.1 Use Case Diagrams for Car Rental System

Use case Request reservation

Use case name: Request reservation

Description: The customer requests the car rental clerk for reservation

Actors: Customer

Goal: To request car rental reservation

Precondition: Customer requested reservation

Postcondition: Request rejected; Request accepted

Triggering event: A customer requests a reservation

Extensions:

Alternatives:

Use case Request rental extension

Use case name: Request rental extension

Description: The customer requests the car rental clerk for rental extension

Actors: Customer

Goal: To request car rental extension

Precondition: Customer wants to extend rental

Postcondition: Request rejected; Request accepted

Triggering event: A customer requests rental extension
Extensions:
Alternatives:

Use case **Request cancel reservation**

Use case name: Request cancels reservation
Description: The customer requests the car rental clerk for canceling reservation
Actors: Customer
Goal: To request cancel reservation
Precondition: Customer has reservation and requested cancellation
Postcondition: Request rejected; Request accepted
Triggering event: a customer requests cancel reservation
Extensions:
Alternatives:

Use case **Pay**

Use case name: Pay
Description: The customer pays charge of the rental by credit card or cash
Actors: Customer
Goal: To enable customer to pickup the car
Precondition: The reservation extension of the rental or extra charge to be paid is present or exists
Postcondition: The rental charge is paid; the rental charge is not paid
Triggering event: Customer wants to pickup a car, or extends the rental.
Extensions:
Alternatives:

Use case **Pickup**

Use case name: Pickup a car
Description: The customer wants to pickup the car that already he/she has reserved
Actors: Customer
Goal: To enable customer use the car
Precondition: The reservation of the rental to be paid is present or exists and the payment is done
Postcondition: The car picked up by the customer
Triggering event: customer wants to pickup a car
Extensions:
Alternatives:

Use case **Return**

Use case name: Return
Description: The customer wants to return a car for a car rental
Actors: Customer
Goal: To return a car for a car rental and in order to prevent that; a customer must pay extra charge for late return
Precondition: Current reservation exists and a car has already been picked up by customer or (car has been delivered to the customer)
Postcondition: A car is returned back to a car rental company
Triggering event: A customer requests a return on the end day of the rental
Extensions:
Alternatives:

Use case **Reply to customer requests**

Use case name: Reply to customer requests
Description: The Car rental clerk replies to customer requests
Actors: Car rental clerk
Goal: To provide services to the customer
Precondition: Customer requests a specific service
Postcondition: The customer replied to
Triggering event: A customer requests a reservation; customer requests a cancellation; customer requests extending the rentals.
Extensions:
Alternatives:

Use case **Handle reservation**

Use case name: Handle reservation

Description: The car rental clerk handles the reservation that takes place

Actors: Car rental clerk

Goal: To achieve the reservation process

Precondition: The reservation requested by the customer; the cancellation requested by the customer

Postcondition: The reservation request dealt with

Triggering event: The customer requests reservation.

Extensions: Allocate car for customer, reserve, cancel, or eject reservation

Alternatives:

Use case **Reserve**

Use case name: Reserve

Description: the car rental clerk reserves a car of a specific category for specific customer

Actors: Car rental clerk

Goal: To reserve a specific car for the customer

Precondition: Customer requested reservation

Postcondition: A new reservation exists; specific car is reserved for the customer

Triggering event: The customer requests the clerk to reserve a car for him/her

Extensions: check blacklist, verify rules and check customer demands

Alternatives:

Use case **Cancel**

Use case name: Cancel

Description: The car rental clerk cancels a reservation that already exists

Actors: Car rental clerk

Goal: To prevent the picking up of a car for which a reservation was made

Precondition: The reservation exists

Postcondition: The reservation is marked as cancelled; no car will be picked up for this reservation.

Triggering event: A customer requests the clerk to cancel a reservation

Extensions:

Alternatives:

Use case **Reject reservation**

Use case name: Reject reservation

Description: The car rental clerk rejects the reservation

Actors: Car rental clerk

Goal: To prevent the rentals which against car rental rules

Precondition: The customer request is made or the customer does not fit to the car rental rules

Postcondition: The reservation is rejected

Triggering event: Car rental rules not satisfied

Extensions:

Alternatives:

Use case **Handle rental**

Use case name: Handle rental

Description: The car rental clerk handles the rental that takes place

Actors: Car rental clerk

Goal: To achieve the rental transaction

Precondition: The rental to be handled is present

Postcondition: The rental transaction is achieved

Triggering event: The customer ready to pay the rental

Extensions: Create rental contract, reject rental, and calculate price

Alternatives:

Use case **Extend rental**

Use case name: Extend rental

Description: The car rental clerk extends a rental that is already in effect

Actors: Car rental clerk

Goal: To prevent the situation where a car must be returned back to the rental company

Precondition: The rental to be extended does exist

Postcondition: The rental is marked as extended; the car will be returned back on extended date
Triggering event: A customer requests the car rental clerk to extend a rental
Extensions:
Alternatives:

Use case **prepare rental contract**

Use case name: Prepare rental contract
Description: The car rental clerk prepares contract for the reservation that has already been done
Actors: Car rental clerk
Goal: To preserve assets of car rental company
Precondition: The reservation to be contracted exists
Postcondition: The contract is marked as confirmed and accomplished
Triggering event: A customer confirms and decides to rent a car
Extensions:
Alternatives:

Use case **Calculate price**

Use case name: Calculate price
Description: The car rental clerk calculates the total price of the rental
Actors: Car rental clerk
Goal: To provide a customer with the total price
Precondition: The existence of the rental for which the total price is to be calculated
Postcondition: The total price of the rental is calculated
Triggering event: A customer confirms and decides to rent a car
Extensions: Handover
Alternatives:

Use case **Handover**

Use case name: Handover
Description: The car clerk delivers the car to customers
Actors: Car clerk
Goal: To check the car whether it is already has damage or not
Precondition: The customer asks to pickup the car
Postcondition: The car delivered to customer
Triggering event: A customer is ready to pickup the car
Extensions: Check car, prepare
Alternatives:

Use case **Check car**

Use case name: Check car
Description: The car clerk checks the car when it comes back to the garage
Actors: Car clerk
Goal: To know whether the car has any damage or not
Precondition: The car to be checked is present
Postcondition: The car is checked and ready for rent
Triggering event: The customer has returned the car back to the company garage and it is already
in
Extensions:
Alternatives:

Use case **Prepare car**

Use-case name: Prepare car
Description: The car clerk prepares the car to be ready for rental
Actors: Car clerk
Goal: To satisfy customer demands
Precondition: A customer requests the car rental clerk to pickup a car
Postcondition: The car is prepared and ready to be picked up by the customer
Triggering event: The car comes back to the garage
Extensions:
Alternatives:

Use case **Reject rental**

Use case name: Reject rental
Description: The car rental clerk rejects the rental
Actors: Car rental clerk
Goal: To prevent the picking up of a car that has been reserved
Precondition: The customer is not able or is refusing to pay the rental
Postcondition: The rental is rejected; no car will be picked up for this reservation
Triggering event: A rental payment is failed
Extensions: -
Alternatives:

Use case **Handle car service**

Use case name: Handle car service
Description: The car rental clerk manages car service
Actors: Car rental clerk
Goal: To ensure that the cars are ready for rent
Precondition: The service date for the car is due
Postcondition: The car is serviced and ready for rent
Triggering event: A car needs to be serviced
Extensions:
Alternatives:

6.3.1.2 UCMs for the Car Rental System

The next step of the system scenario model is developing UCMs for the car rental system. In this step, use case maps of the car rental system are described. UCMs are applied in order to capture the behavior of the car rental system in high level description and explain how UCMs describe the system scenario in visual views. UCMs discover car rental system roles, and responsibilities along the way. In this step, we should perform the following important tasks:

1. Identify scenarios and major components involved in the car rental system.
2. Identify roles for each component.
3. Identify preconditions and postconditions for each scenario.
4. Identify responsibilities and constraints for each component in a scenario.
5. Identify sub scenarios and replace them with stubs.
6. Identify components collaborations for the major tasks.

The car rental system is decomposed into the sub-scenarios as follows: reservation scenario, car pickup scenario, car return scenario, rental extension scenario and car service scenario. Each scenario is described in detail.

6.3.1.2.1 Reservation Scenario

The reservation scenario will be performed between two components of the system called: customer and car rental clerk. Figure 6.2 shows the use case map for the reservation scenario.

The customer component represents the customer in the application environment, and the car rental clerk represents the employee of the car rental company. The preconditions for the reservation scenario are:

- A customer wants to rent a car.
- A reservation is already done and the customer wants to cancel the reservation.

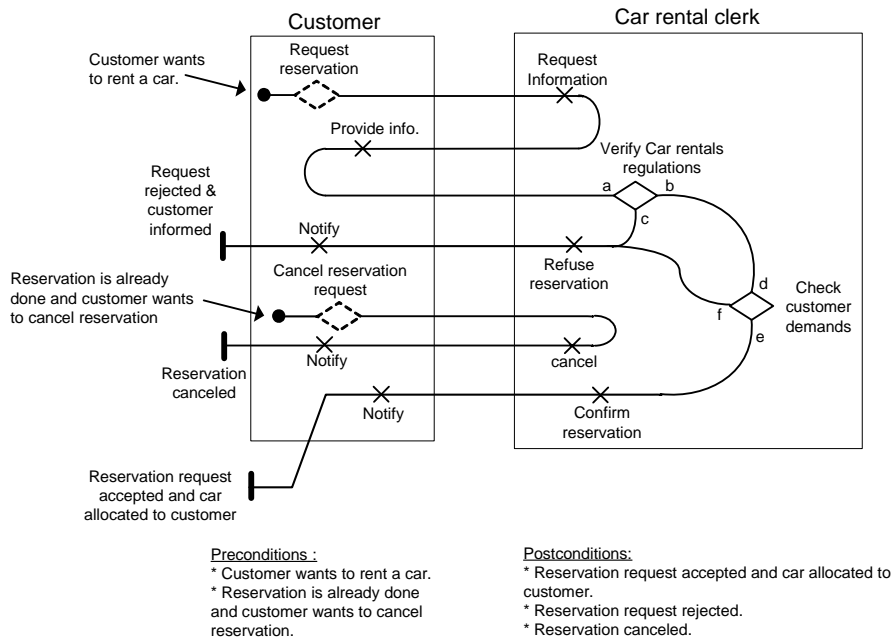


Figure 6.2 Use Case Map for Reservation Scenario

When the first precondition is satisfied the scenario starts with the *Request reservation* stub, which hides the detailed information of the *request reservation* process. The request reservation can be achieved in several ways. For example, it can be done by a phone call, or by filling a web form, or by an Email. Therefore, the *request reservation* stub is represented as a dynamic stub. Figure 6.3 illustrates the plug-ins for the *request reservation* dynamic stub. After all responsibilities for the *request reservation* process are performed, the path leads to the car rental clerk component. In this component there is a responsibility called *request information*, which requests the customer to provide his/her personal information such as address, phone, personal ID, driving license etc. The path leads to the customer component where there is a responsibility called *provide info*, which provides a confirmation that the customer has filled in the application form for the rental transaction.

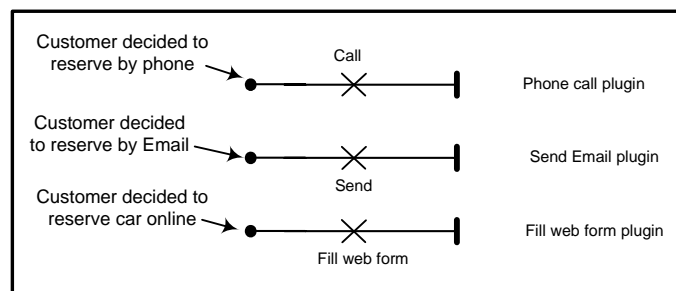


Figure 6.3 Plug-Ins for Request Reservation Stub

After the previous responsibilities are performed, the path leads to the static stub *verify car rentals regulations* which hides the detailed information of the *verify car rentals regulations* process. This stub should be achieved in one specific mode.

Figure 6.4 illustrates the plug-in for the *verify car rentals regulations*. In this plug-in the car rental clerk checks whether the customer meets the rental rules of the car rentals company. These regulations are represented by the following tasks or responsibilities (*verify rules*, *check blacklist* and *check simultaneous reservations*) that should be performed by the clerk of the car rentals company. The path starts with the *verify rules* responsibility

which verifies the rules of the car rentals company such as customer age, validity of drivers license etc. Then the path leads to an or-fork immediately after the *verify rules* responsibility, which indicates alternative scenario paths.

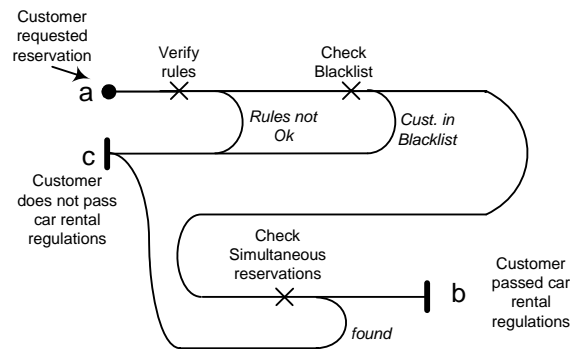


Figure 6.4 Plug-In for Verify Car Rentals Regulations

One path leads to the end point “c” which means the *reservation request* is rejected, e.g. because the customer does not have a valid drivers license. The other path leads to the next responsibility *check blacklist*. The *check blacklist* responsibility checks whether the customer belongs to the customers blacklist or not. In the same situation the path leads to an or-fork immediately, which indicates alternative scenario paths. One path leads to the end point “c” which means the reservation request is rejected, since the customer is included in the blacklist. The other path confirms that the customer is not included in the blacklist, leading to the last responsibility *check simultaneous reservations*. It checks whether the customer has reservation for more than one car at a time. A customer may have multiple future reservations, but may have only one car at any time. After the *check simultaneous reservations* responsibility is checked the path leads immediately to an or-fork, which indicates alternative scenario paths. One path leads to the end point “c” which means the reservation request is rejected, since the customer already has another car, which according to the car rentals rules is not allowed. The other path leads to the end point “b” which confirms that customer passed the car rentals regulations and he/she is allowed to reserve a car.

The *verify car rentals regulations* stub has two outgoing ports. If the customer passed the car rentals regulations, port “b” will be followed, which means that the customer is allowed to reserve a car. Otherwise, port “c” is followed, which means that the customer reservation request is rejected. The path that comes from port “b” leads to the *check customer demands* stub, which hides the detailed information of the *check customer demands* process. This stub checks whether the customer demands are available or not.

Figure 6.5 illustrates the plug-ins for the *check customer demands* stub. In this plug-in, the car rental clerk checks the customer’s demands. This plug-in is represented by the following tasks or responsibilities: check availability of customer demands, check customer demands in other branches, propose another car, assess, and verify. The path is started with the responsibility *check availability of customer demands* which checks whether the customer’s demands are available in this branch or not.

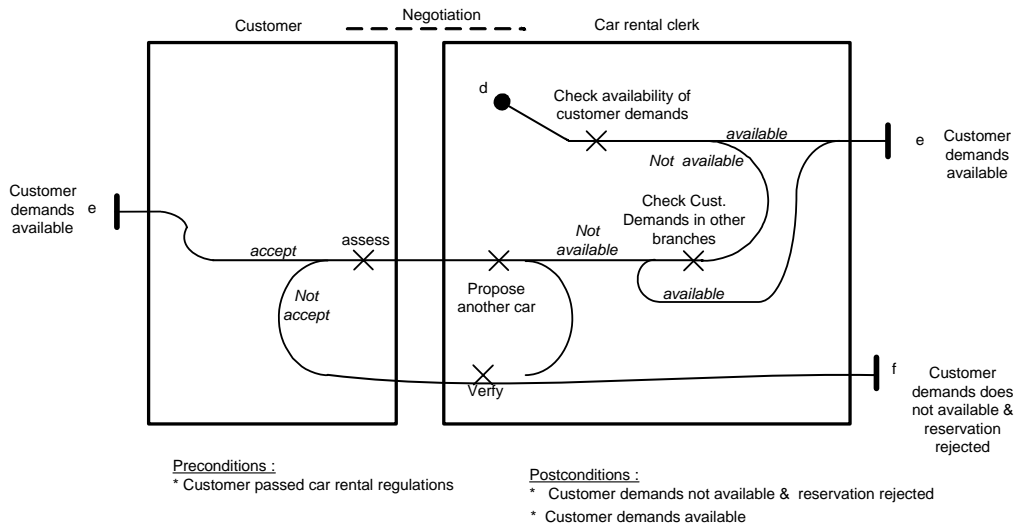


Figure 6.5 Plug-In for the Check Customer Demands Stub

After that, check *availability of customer demands* responsibility invoked, the path leads to an or-fork, which indicates alternative scenario paths. One path labeled *available* leads to the end point “e” which means the customer demands are available. The other path (labeled *Not Available*) leads to the responsibility *check customer demands in other branches*. This responsibility finds out whether the customer demands are available in other branches or not.

After *check customer demands in other branches* responsibility is checked, the path immediately leads to an or-fork, which indicates alternative scenario paths. One path labeled *available* leads to the end point “e” which means the customer demands are available in some other branch. The other path (labeled *not available*) leads to responsibility *propose* which proposes to the customer another car from the same group. The path after that leads to the responsibility *assess*, which confirms that the customer estimates the proposal. Then the path leads to an or-fork. One path labeled *accept*, leads to the end point “e”.

That is an indication that the customer accepts the proposal. The other path (labeled *not accepted*) leads to the responsibility *verify* which verifies that the customer has responded. If the customer asks for another offer the path leads to the responsibility *propose* again. Otherwise, the path leads to the end point “f” which means that the customer demands were not available and the customer reservation is rejected.

The *check customer demands* stub should be returned back to the reservation scenario either by the “e” or “f” port. The path that comes from port “e” leads to the responsibility *confirm reservation* and then leads to the end point *reservation request accepted and car allocated to customer*. The path that comes from the port “f” leads to the responsibility *refuse reservation* and then the path leads to the end point *request rejected and customer informed*.

When the second precondition of the reservation scenario is satisfied; the scenario starts with the *cancel reservation request* stub, which hides the detailed information of the *cancel reservation request* process. The *cancel reservation request* stub can be achieved in

several ways. For example, it can be done through a phone call, by filling a web form, or by an Email. Figure 6.6 illustrates the plug-ins for the *cancel reservation request* stub. After all responsibilities for the *cancel reservation request* process are performed, the path leads to the car rental clerk component where there is a responsibility called *cancel*, which cancels the reservation that is already done by the customer. Then the path leads to a responsibility *confirm* which confirms that the reservation is canceled. Then the path leads to the customer component where there is a responsibility called *receive confirmation*, which indicates that the confirmation for cancellation is received by the customer. Then the path leads to the end point *reservation canceled*.

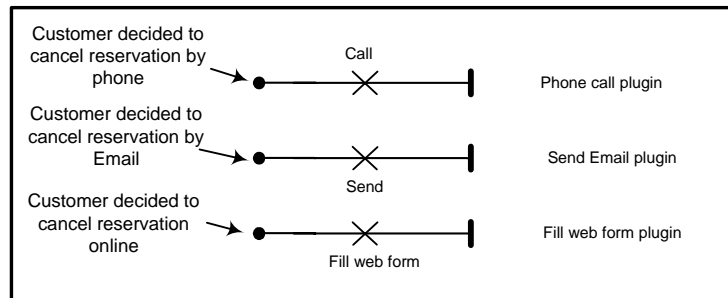


Figure 6.6 Plug-Ins for Cancel Reservation Request Stub

6.3.1.2.2 Car Pickup Scenario

The car pickup scenario will be performed between the customer and car rental clerk components of the system. Figure 6.7 shows the pickup scenario of UCMs. The precondition for the pickup scenario is that *reservation already done* and *car allocated to the customer*. The scenario starts with responsibility *request to pickup car* stub, which indicates that the customer wants to pickup the car based upon his/her reservation.

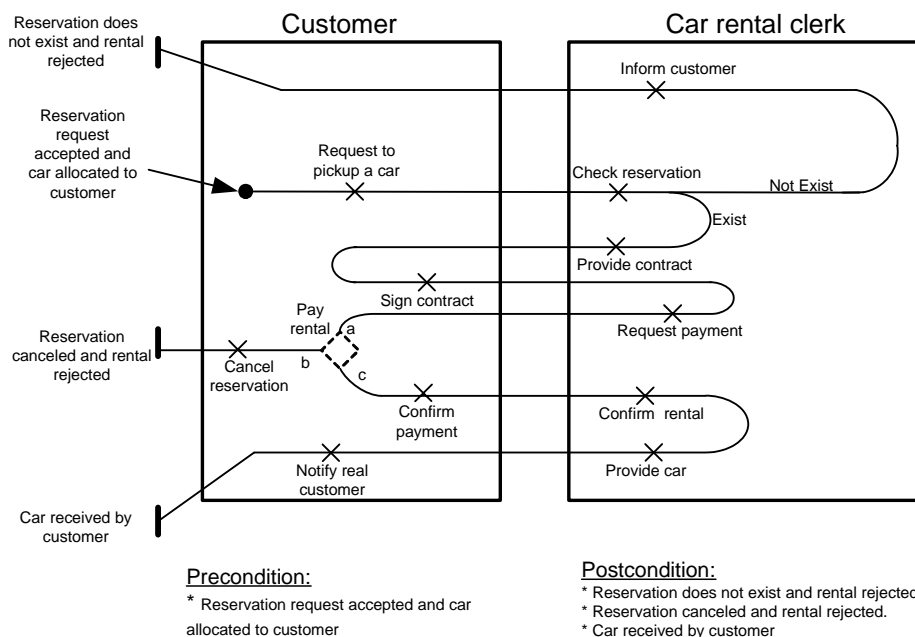


Figure 6.7 Car Pickup Scenario

The path goes to the car rental clerk component and leads to the responsibility *check reservation* in which a check is made whether the customer has a reservation or not. Then the path leads to an or-fork immediately, which indicates alternative scenario paths. One path labeled *not exist* leads to the responsibility *inform customer*, which informs the customer that he/she has no reservation. The path subsequently leads to the end point *reservation does not exist and reservation request rejected*, which is considered as a post-condition of this scenario. The path labeled *exist* leads to responsibility *provide contract*, which is responsible for providing the contract to be signed by the customer. Then, the path goes to the customer component and leads to responsibility *sign contract*, which is a commitment by the customer to sign the contract. The path then goes to the car rental component to the responsibility *request payment*, where the path goes to the customer component and leads to the *pay* stub, which hides the detailed information of the payment process. This stub shows the payment process details. Figure 6.8 illustrates the plug-ins for the *pay rental* stub

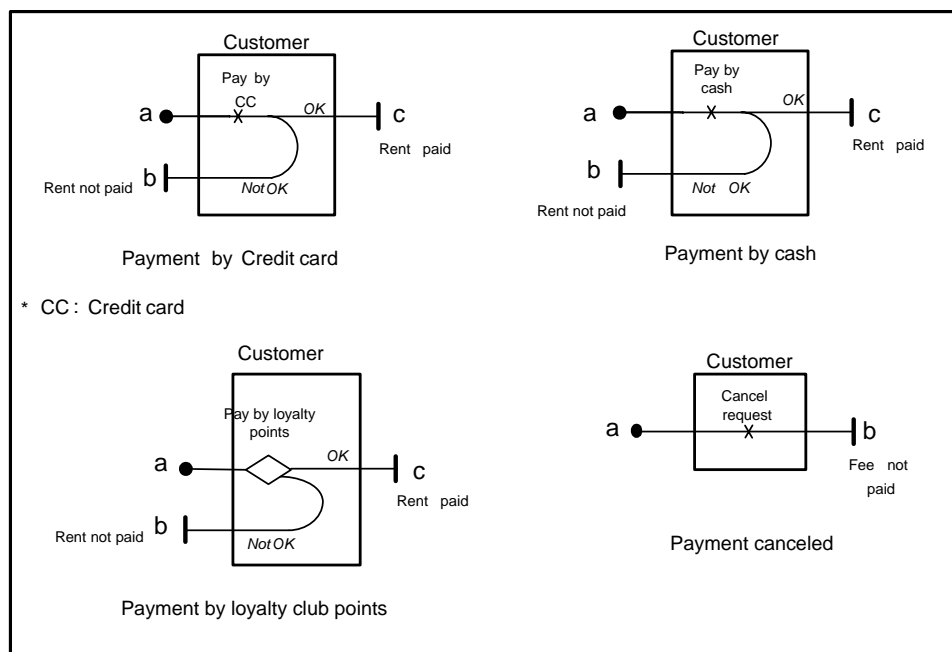


Figure 6.8 Plug-ins for Pay Rental Stub

The *pay* stub has two outgoing ports. If the customer has not paid the rentals fees, then port *b* will be followed, which means that the customer does not pay the fee. This path leads to responsibility *cancel reservation*. It then leads to the end point *reservation cancelled and rental rejected* which is considered as a post condition of this scenario. Otherwise, port *c* is followed, which means that the customer has paid the rentals fees. The path that comes from port “c” leads to the responsibility *confirm payment*. After that, the path leads to the responsibilities *confirm rental* and *provide car*, which is a commitment by the company to provide the car to the customer. Subsequently, the path leads to the responsibility *notify real customer* at customer component to pickup the car. It then goes to the end point *Car received by customer*, which is considered a post-condition of this scenario.

Figure 6.9 illustrates the plug-in of this *pay by loyalty points* stub. This plug-in starts when the precondition *Customer used loyalty points for payment* is satisfied. The path starts

with the responsibility *Deliver membership* that confirms that the customer has delivered his/her loyalty club membership. Then the path goes to the car rentals clerk and leads to the responsibility *Check customer membership* which checks whether the customer has loyalty club membership or not. Then the path immediately leads to an or-fork, which indicates alternative scenario paths.

One path labeled *No* leads to the responsibility *payment request*, which informs the customer that he/she has to pay in cash or use a credit card. The path then leads to the end point *Loyalty points not enough and payment requested* which is considered as postcondition of this plug-in.

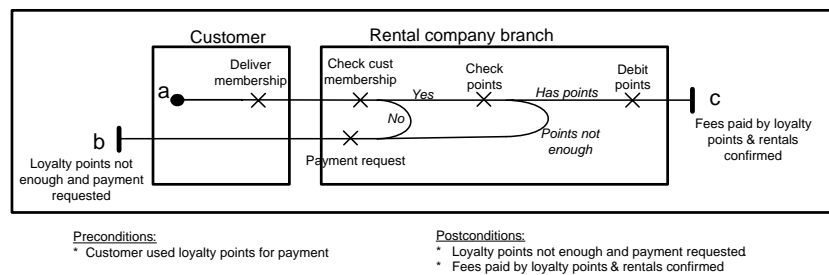


Figure 6.9 Plug-Ins for Pay by Loyalty Points Stub

The other path leads to the responsibility *check points* which checks whether the customer has enough points to be able to pay by loyalty points or not. Then the path immediately leads to an or-fork, which indicates alternative scenario paths. One path labeled *points not enough* leads to the responsibility *payment request*, which informs the customer that he/she has to pay in cash or by use a credit card.

The path then leads to the end point *Loyalty points not enough and payment requested* which is considered as post-condition of this scenario. The other path labeled *has points* leads to the responsibility *debit points* which debit points from the account of the customer loyalty club. The path then leads to the end point *fee paid by loyalty points and rentals confirmed* which is considered as post-condition of this plug-in.

6.3.1.2.3 Car Return Scenario

The car return scenario will be performed between the customer and the rental company clerk components of the system. Figure 6.10 shows the car return scenario of UCMs. The preconditions for the rentals scenario are that *customer wants to return a car* and *car is damaged*.

This scenario starts either with the precondition *customer wants to return a car* or with the precondition *car is damaged*. When the precondition *car is damaged* is satisfied then the scenario starts with responsibility *inform car rental*, which informs the Car Rentals Company that the car is damaged. Then the path goes to the car rental clerk component and leads to responsibility *receive a car*. When the precondition *customer wants to return a car* is satisfied then the scenario starts with the responsibility *return car*, which confirms that the customer has returned the car to company garage.

After that, the path goes to the clerk component and leads to the responsibility *receive car* where it indicates that the rental company clerk has received the car. The path then leads to *terminate transaction* stub which hides the detailed information of the *terminate transaction* process. This stub should be achieved in only one specific way. Therefore, the *terminate transaction* stub is represented as a static stub.

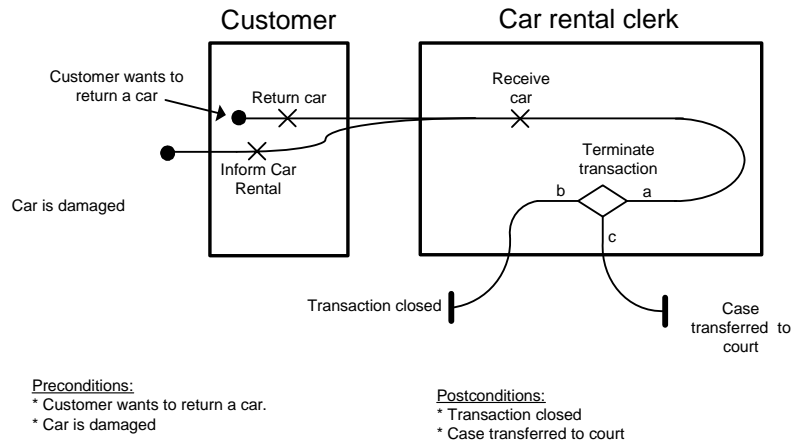


Figure 6.10 UCM for Car Return Scenario

Figure 6.11 illustrates the plug-in for the *terminate transaction* stub. This plug-in starts with *check return date* responsibility. This responsibility checks whether the car has been returned on time or not. Then the path immediately leads to an or-fork after the *check return date* responsibility, which indicates alternative scenario paths. One path labeled *on time* indicates whether the car was returned on time and the path then leads to responsibility *check car condition*.

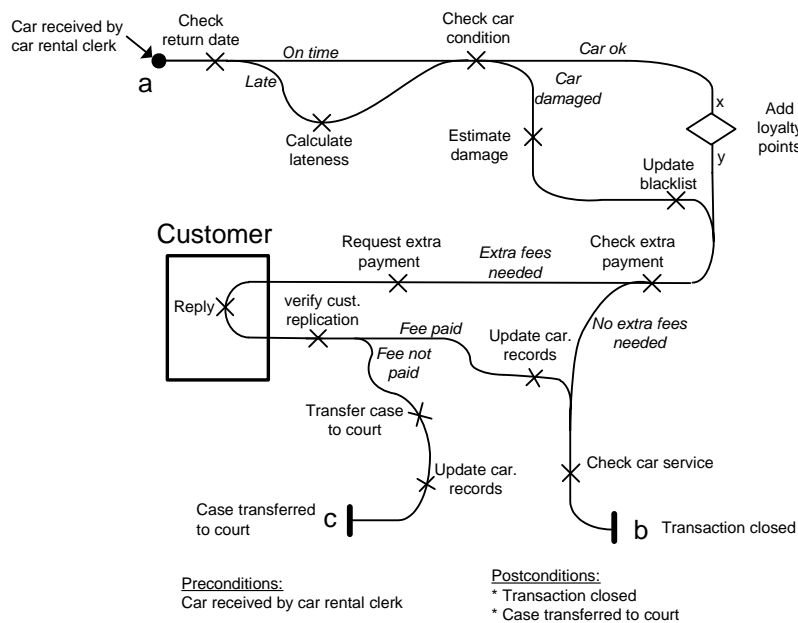


Figure 6.11 Plug-Ins for the Terminate Transaction Stub

The other path labeled *late* indicates that the car was delayed beyond the proper time of arrival which leads to the responsibility *calculate lateness*. This calculates the cost of how late the car was as an extra payment. The path then leads to the responsibility *check car condition*. The responsibility *check car condition* verifies whether the returned car was damaged or not. The path then immediately leads to an or-fork, which indicates alternative scenario paths. One path labeled *car damaged* leads to two successive responsibilities *estimate damage* and *update blacklist*. The *estimate damage* responsibility is concerned with estimating the car damage. The *update blacklist* responsibility is concerned with updating the blacklist by adding this customer to the list. The path then leads to an or-joint that leads to the responsibility *check extra payment*.

The other path labeled *car ok* leads to the *add loyalty points* stub which hides the detailed information of the *add loyalty points* process. Figure 6.12 illustrates the plug-in of the *add loyalty points* stub.

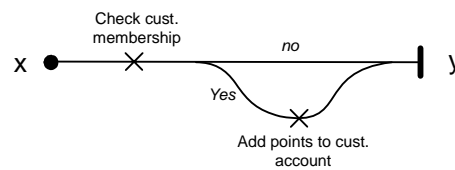


Figure 6.12 Plug-In for Add Loyalty Points Stub

The *add loyalty points* plug-in starts with the *check customer membership* responsibility which is responsible for checking the customer membership of the loyalty club. Then the path leads to an or-fork immediately after the *check customer membership* responsibility, which indicates alternative scenario paths. One path is labeled *no* which indicates that the customer is not a member of the loyalty club and the path then leads to the end point “y”. The other path is labeled *yes*, which means the customer is a member of loyalty club and then the path then leads to the responsibility *add points to customer account* which adds points to the customer account of the loyalty club. After that, the path leads to the end point “y”. The path leads to or-joint which leads to the responsibility *check extra payment*.

The responsibility *check extra payment* in the terminate transaction plug-in checks whether the customer should pay an extra fee or not. The path leads to an or-fork immediately, which indicates two alternative scenario paths. One path labeled *no extra fee needed* leads to responsibility *check car services* which checks whether the car needs service or not. Then the path leads to the end point “b”.

The other path labeled *extra fees needed* leads to the responsibility *extra payment request*. In the case of a delayed car return, or the car is damaged, an extra payment request responsibility requests the customer to pay an extra fee. The path then goes to the customer component and leads to the responsibility *reply*, which confirms that the customer has replied to the car rental company. The path goes back to the car rental clerk component and leads to the responsibility *verify customer replication* that verifies whether the customer has paid the extra fee or not. Then the path immediately leads to an or-fork, which indicates alternative scenario paths.

The path labeled *fee paid* leads to the responsibility *update car records*. The other path labeled *fee not paid* leads to the responsibility *transfer case to court*. This means that the case is transferred to the court because of the customer’s lack of payment of the extra

fee. After the responsibility *transfer case to court* has been performed, the path then immediately leads to an and-fork, which has two simultaneous scenario paths. One path leads to the end point “c”. The other path leads to responsibility *update car records*.

The terminate transaction stub leads to two ports “b” and “c”. The path that comes from port “b” indicates that the transaction is closed. The path that comes from port “c” indicates that the transaction has been transferred to court.

6.3.1.2.4 Rental Extension Scenario

The Rental extension scenario will be performed between the customer and the car rental clerk components of the system. Figure 6.13 shows UCMs of the rental extension scenario.

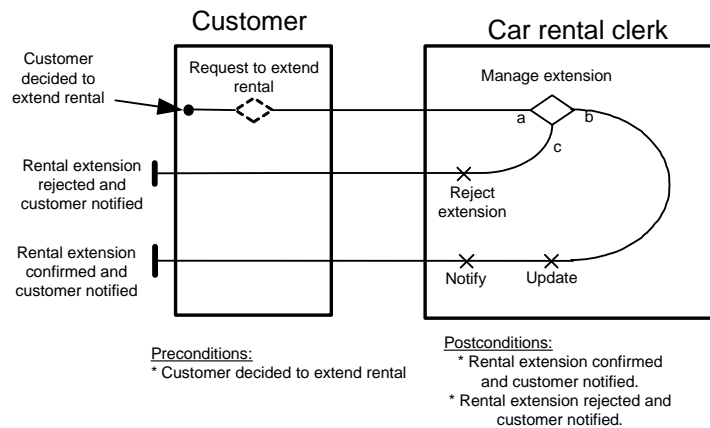


Figure 6.13 UCM for Rental Extension Scenario

The precondition for the rental extension scenario is that the *customer decided to extend rental*. The scenario starts with the *request to extend rental* stub, which hides the detailed information of the *request to extend rental* process.

The *request to extend rental* can be achieved in several ways. For example, it can be done through a phone call, or by filling a web form, or by an Email. Figure 6.14 illustrates the plug-ins for the *request to extend rental* stub. After the *request to extend rental* stub is performed, the path goes to the car rental component and leads to the *manage extension* stub. This in turn hides the detailed information of the *manage extension* process.

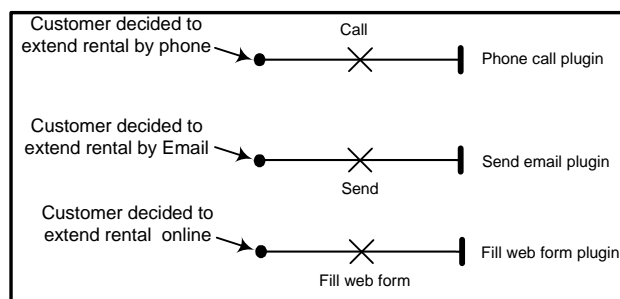


Figure 6.14 Plug-Ins for Request to Extend Rental Stub

Figure 6.15 shows the plug-in of the *manage extension* stub. This plug-in starts with the start point “a” where the path leads to the *check service date* which checks whether

the service date of the car is due. After the responsibility *check service date* has been performed, the path immediately leads to an or-fork, which indicates two alternative paths.

One path is labeled *no service required* which means the car does not need a service. The customer is allowed to extend his/her rental and the path leads to the end point “b”. The other path labeled *service required* immediately leads to the responsibility *propose another car* then the path moves to the customer component and leads to the responsibility *reply* which means that the customer provides a reply for the proposed car. The path then moves back to the clerk component and leads to responsibility *verify*, which verifies whether the proposal is accepted or not. The path then leads to an or-fork immediately, which indicates two alternative paths. One path labeled *proposal not accepted* leads to the end point “c” which means the rental extension is not accepted and the car should be given back to the car rental. The other path labeled *proposal accepted* leads to the end point “b” which means the rental extension is accepted.

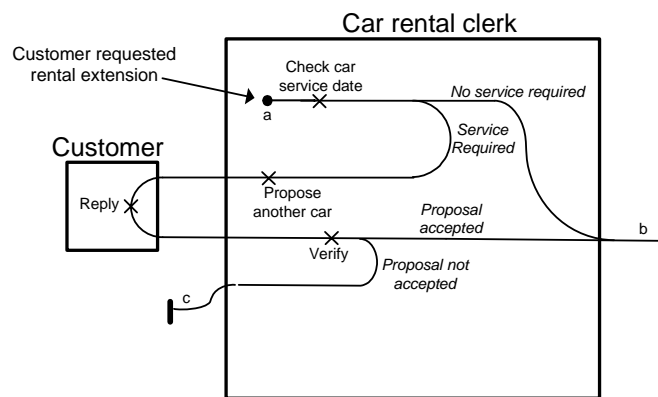


Figure 6.15 Plug-ins for Manage Extension Stub

The *manage extension* stub should return to the reservation scenario by either port “c” or “b”. The path that comes from port “c” leads to the end point *rental extension rejected and customer notified*. The path that comes from port “b” leads to the responsibility *update*, which means that the clerk updates the car rental database with the extension. This is followed by the path leading to the responsibility *notify*, which notifies the customer that the extension has been confirmed. The path then leads to the end point *rental extension confirmed and customer notified*.

6.3.1.2.5 Car Service Scenario

The car service scenario will be performed between a car rental clerk and the service clerk components of the system. Figure 6.16 shows UCMs for car service scenario.

Once the precondition *service plan required* for the car service scenario is satisfied, the scenario starts with the *request service reservation* responsibility, which requests a service reservation for a specific car from the service clerk in the service depot. The path moves to the service clerk component and leads to the responsibility *reserve date*, which reserves a date for a car service. Then the path returns to the car rental clerk and leads to the responsibility *accept* which means that car rental clerk has accepted the date. After the responsibility *accept* has been done the path leads to an and-fork

immediately, which splits into two simultaneous paths. One path leads to the end point *service date allocated for car*, which is considered as a postcondition for this scenario. The other path leads to the timer point, which waits until the confirmation report comes from the computer component indicating that the car should be taken to the service depot. Subsequently, the path leads to the responsibility *handover the car*, which means that the car has been taken to service depot. The path moves to the service clerk component and leads to the responsibility *perform service*, which means the service clerk is performing the service. When this responsibility is completed, the path then leads to the end point *service performed and the car is ready*. This is considered as post-condition of this scenario.

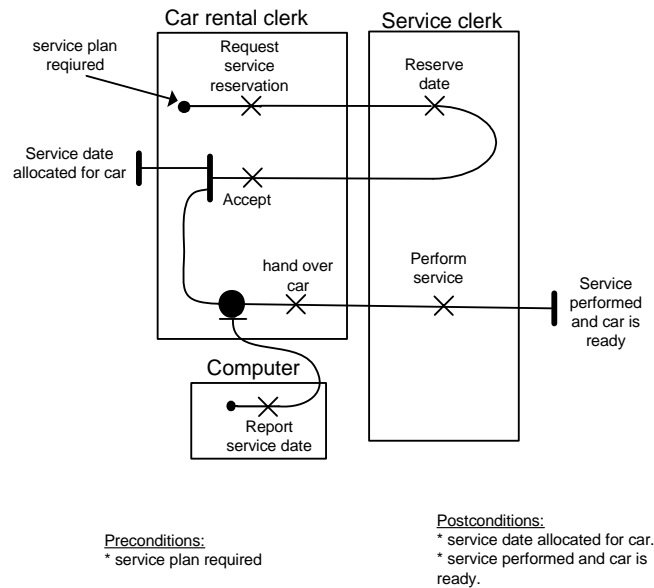


Figure 6.16 UCM Car Service Scenarios

6.4 Analysis Phase

The objective of the analysis phase is to transform the car rental system requirements into a representation of the system that can be forwarded to the design phase.

This phase starts with analyzing the car rental system requirements. It utilizes the system scenario model that is constructed by use case maps and UML use cases. This model is considered as a foundation to produce the models of the analysis phase. The analysis process involves two main stages: agent architecture and MAS architecture.

6.4.1 Agent Architecture Stage

The agent architecture stage is concerned with the following steps:

- 1) Constructing both models: a roles model and an agent model consecutively.
- 2) Identifying system agents and assign roles to them.
- 3) Refining the roles.
- 4) Identifying agent goals and specifying the plans for each goal.
- 5) Model triggers: At the end of agent architecture stage the triggers model is constructed.

6.4.1.1 Roles Model

The roles model is the first model of the analysis phase. It describes how the roles that will be played in the car rental system are discovered.

6.4.1.1.1 Discovering Roles

In this step, we want to determine what roles are played by each component in the proposed car rental system. It will be necessary to trace the existing paths in all system scenario associated with the customer component and car rental clerk component. It will also be necessary to identify all UCM responsibilities and stubs, which in turn, will help to obtain the responsibilities of the role. This, consequentially, will make it possible to determine the role of each component in the system.

By looking at the customer component of the reservation scenario, and tracing the paths inside it, we observe the customer performing the *reservation request* stub. In this stub, the customer requests the reservation of a car from the car rental clerk. Then we may need to trace out the sub-scenarios for each stub in order to find out more information that may help to identify and define the required role. This matter takes place especially when the name of the stub not sufficiently accurate to specify the role.

Also, when we are looking at the customer component in the *car pickup* scenario in figure 6.7, it becomes clear that the customer component performs several operations such as request to pickup a car, pay rental, cancel reservation, and sign rental contract. When we look at the customer component in the *car return* scenario in figure 6.10, it turns out that the customer also performs a responsibility called *return car*. This responsibly includes returning a car to the company's garage. The customer component also performs another responsibility called *reply*, which is the response of the customer to fees payment in case the company requests other expenses. In addition, when looking at the *rental extension* scenario, we discovered that the customer performs several operations too. These operations include *rental extension request* where the customer requests to extend the rental of a car he/she already has. The operations the customer performs also include the *reply* responsibility, which represents the customer's response to accept or reject an offer proposed by the company. The offer will consist of an alternative car to be given to the customer in the case that the car that the customer has at the time, must be returned back to the company for the purpose of maintenance. Once all responsibilities and stubs that the component customer performs have been recognized, it is quite possible to define and specify the role played by the component customer. There are two roles that are played in the car rental system: renter and rentier.

6.4.1.1.2 Roles of the Car Rental System

The roles model exhibits all the roles that will be performed by the customer and the car rental clerk components. This model includes responsibilities for each role and the activities for each responsibility. Table 6.1 illustrates the renter role for the customer component. How to build these models have been addressed in detail in Chapter 5.

Role name:	Renter
Role description:	Renter who pays rent to use a car that is owned by the car rental company.
Responsibilities & its Activities:	Res 1. Request reservation. Act1. Reserve car by a phone call. Act2. Reserve car by e-mail. Act4. Reserve car by the Internet. Res 2. Cancel reservation request. Act1. Cancel reservation by a phone call. Act2. Cancel reservation by e-mail. Act3. Cancel reservation by the Internet. Res 3. Request to pickup a car. Act1. Request to pickup a car. Res 4. Sign contract. Act1. Read and Sign the rental contract. Act2. Pay rental. Act3. Pay rentals by cash. Act4. Pay rentals by a credit card. Act5. Pay rentals by loyalty club points. Res 5. Pay extra charge Act1. Pay for damage costs by cash. Act2. Pay for damage costs by credit card. Act3. Pay for late return by cash. Act4. Pay for late return by credit card Res 6. Notify real customer Act1. Notify customer for picking up a car Act2. Remind customer about return date Act3. Notify customer for canceled reservations Act4. Notify customer for rejected reservations Act5. Notify customer for confirmed reservations Res 7. Return car Act1. Return car to the company garage Res 8. Extend Rental Act1. Request for extending rentals Act2. Receive confirmation for car rentals Act3. Negotiate car rentals proposals Act4. Notify customer for rejecting extending rentals Act5. Notify customer for accepting extending rentals
Obligations:	<ul style="list-style-type: none"> • Renter should pay insurance for each car rental • Contact car Rental Company in urgent cases • Renter should not leave the car when the car is damaged until the car rentals company receives it • The renter should pass rental regulations
Permissions:	<ul style="list-style-type: none"> • Servicing the car in Urgent cases • Authorize another driver for a car
Constrains:	<ul style="list-style-type: none"> • The renter must have a valid driver's license • The car must be insured • The renter should not visit the countries that the insurance does not cover • The car should only be driven by the renter • The driver must be over 25 • The renter should not have more than one reservation at the same time

Table 6.1 Renter Role for Customer Component

Table 6.2 illustrates the rentier role for the car rental clerk component.

Role name:	Rentier
Role description:	Rentier who is renting cars to customers.
Responsibilities & its Activities:	Res 1. Make reservation. Act1. Request information. Act2. Verify car rentals' regulations Act3. Check Customer's demands Act4. Confirm reservation. Act5. Reject reservation Res 2. Cancel reservation Act 1. Cancel a reservation. Res 3. Handle cars service. Act1. Book an appointment for a car service. Act2. Receive a car service date. Res 4. Allocate cars to customers. Act1. Allocate a car to a customer. Act2. Allocate a car to a customer from another branch. Act3. Allocate a car to a customer from another car rentals company. Res 5. Reply customer requests. Act1. Reply to a customer's reservation request. Act2. Reply to a customer's cancellation request. Act3. Negotiate with a customer. Res 6. Handle rental transaction. Act1. Prepare the rentals contract. Act2. Obtain insurance for a car to be rented. Act3. Confirm payment. Act4. Terminate a rental transaction. Res 7. Handle extension. Act1. Manage a rentals extension.
Obligations:	<ul style="list-style-type: none"> • The rentier should satisfy customers' demands. • The rentier should apply car rental rules. • The rentier must be keep cars papers and documents valid. • A car should be insured before given to renter. • Local taxes must be collected on the rental charge. • The rental would not exceed the mileage more than 10% over the normal mileage for the service.
Permissions:	<ul style="list-style-type: none"> • A car scheduled for service may be used. • A car may have to be rented from a competitor.
Constrains:	<ul style="list-style-type: none"> • Rented cars must meet local legal requirements.

Table 6.2 Rentier Role for Car Rental Clerk Component

6.4.1.2 The Agent Model

The agent model is the second model of analysis phase. It is composed of the following detailed steps:

6.4.1.2.1 Identifying Agents for Car Rental System

In this step, every component of UCMs is converted into a particular agent. Each agent is selected based on the role that it will play within the system. The car rental system may include several agents such as a manager agent, a service agent, a car rental clerk agent, a customer agent etc. We are therefore not going to discuss all of them. Two agents were chosen to be explored due to the large amount of communication between them. They are the customer agent and the car rental clerk agent.

In the car rental system, the customer and car rental clerk components are assigned respectively to customer agent and car rental clerk agent. The car rental manager agent represents the branch manager of the car rental company. This agent can play two roles in the system. The first and main role is the director role. The second role is the rentier role. It can play the role of a rentier when there is a considerable need e.g. when many customers crowd the car rental clerk agent at the same time. Fig. 6.17 shows how more than one the role is assigned to one agent.

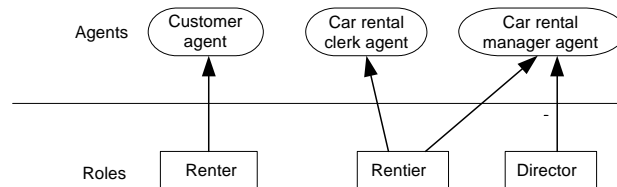


Figure 6.17 Assigning Renter Role to Customer Agent, Rentier Role to Car Rental Clerk and Manager Agents and Director Role to Car Rental Manager Agent.

In the car rental agent system, the role renter is allocated to the customer agent and the role rentier is allocated to the car rental clerk agent. These agents should be able to perform those roles in proper manner.

6.4.1.2.2 Refining Roles

The refining roles step is merely for revising the roles that the agent plays within the system. The refinement process consists of two steps. The first step is to match the roles that are captured in the roles model with agents that play these roles according to the agent's capabilities. The role responsibilities are classified based on who is responsible for performing them. The second step is to separate, or isolate those responsibilities that are to be carried out by real persons from those responsibilities that are to be carried out by agents on their behalf.

The refining roles process concentrates the responsibilities that are to be carried out by the agent. The responsibilities that are to be carried out by real users are stated as preconditions. Agents use these preconditions to keep track of whether the real person performs those responsibilities. These beliefs could be preconditions for other responsibilities. Agents should be able to sense the environment to check whether these beliefs are changed or not. In other words, an agent may wait for a signal (e.g. a message) that confirms that a task performed by the user has been completed.

6.4.1.2.2.1 Refined Roles for Customer Agent

In this example, we illustrate only how the renter role is refined. There is no need to refine the rentier role, because the car rental agent can perform all responsibilities that belong to this role. Therefore, the role will remain as it is. Table 6.3 illustrates the refined renter role for the customer agent. In this model, the responsibility sign contract is removed and stated as the precondition *the contract is signed*. This precondition is used to keep track of whether the real customer signs the contract or not. This condition should be satisfied in order for the customer agent proceeds rental process.

Role name:	Renter
Role description:	Renter who pays rent to use a car that is owned by the car rental company.
Responsibilities & its Activities:	Res.1 Request reservation. Act 1. Reserve car by phone call. Act 2. Reserve car by e-mail. Act 3. Reserve car by Internet. Res 2. Cancel reservation request. Act 1. Cancel reservation by phone call. Act 2. Cancel reservation by Email. Act 3. Cancel reservation by Internet. Res 3. Request to pickup a car. Act 1. Request to pickup a car. Res 4. Pay rental. Act 1. Pay rentals by cash. Act 2. Pay rentals by credit card. Act 3. Pay rentals by loyalty club points. Res 5. Pay extra charge. Act 1. Pay for damage costs by cash. Act 2. Pay for damage costs by credit card. Act 3. Pay for late return by cash. Act 4. Pay for late return by credit card. Res 6. Notify real customer. Act 1. Notify customer for picking up a car. Act 2. Remind customer about return date. Act 3. Notify customer for cancelled reservations. Act 4. Notify customer for rejected reservations. Act 5. Notify customer for confirmed reservations. Res 7. Extend Rental. Act 1. Request for extending rentals. Act 2. Reply to car rentals proposals. Act 3. Notify customer extension rejected. Act 4. Notify customer extension confirmed.
Obligations:	<ul style="list-style-type: none"> • Renter should pay insurance for each car rental • Contact car Rental Company in urgent cases • Renter should not leave the car when the car is damaged until the car rentals company receives it • The renter should pass rental regulations
Permissions:	<ul style="list-style-type: none"> • Servicing the car in Urgent cases • Authorize another driver for a car
Constrains:	<ul style="list-style-type: none"> • The renter must have a valid driver's license • The car must be insured • The renter should not visit the countries that the insurance does not cover • The car should only be driven by the renter • The driver must be over 25 • The renter should not have more than one reservation at the same time

Table 6.3 Refined Renter Role for Customer Agent

6.4.1.3 Beliefs Model

In this model, we describe the beliefs of the customer agent. The following section shows how the customer agent's beliefs are captured.

6.4.1.3.1 Beliefs of Customer Agent

The identification of customer agent beliefs is performed by tracing the path segments of UCM scenarios for the customer component and capturing all pre-

conditions and postconditions. This is followed by the transfer of those pre- and postconditions into beliefs according to a specific goal or a specific plan or both. Furthermore, all obligations, permission, and constraints of the renter role for customer agents are transferred into beliefs. Table 6.4 depicts the beliefs of the customer agent.

Belief	Type	Purpose
Agent Id	Constant	Storage
Customer wants to rent a car	Variable	Storage
Customer decided to reserve by phone	Variable	Storage
Customer decided to reserve by e-mail	Variable	Storage
Customer decided to reserve car online	Variable	Storage
Reservation confirmed	Variable	Storage
Reservation rejected	Variable	Storage
Customer wants to cancel reservation	Variable	Storage
Customer decided to cancel reservation by phone	Variable	Storage
Customer decided to cancel reservation by e-mail	Variable	Storage
Customer decided to cancel reservation online	Variable	Storage
Cancellation confirmed	Variable	Storage
Payment requested by car Rentals Company	Variable	Storage
Customer decided to pay rental cash	Variable	Storage
Customer decided to pay rental by credit card	Variable	Storage
Customer decided to pay rental by loyalty club points	Variable	Storage
Rental fee paid	Variable	Storage
Rental fee not paid	Variable	Storage
Car is damaged	Variable	Storage
Car returned late	Variable	Storage
Customer decided to pay extra charge cash.	Variable	Storage
Customer decided to pay extra charge by credit card	Variable	Storage
Fee paid	Variable	Storage
Fee not paid and transaction transferred to court	Variable	Storage
Real customer must be notified	Variable	Achieve
Reservation already done and car allocated to the customer	Variable	Storage
The car received by customer	Variable	Storage
Cancel reservation is already requested by customer	Variable	Storage
Reservation Request is already done by customer	Variable	Storage
Customer notified	Variable	Storage
Customer confirmed	Variable	Storage
Customer decided to extend rentals	Variable	Storage
Another offer proposed by car rental clerk agent	Variable	Storage
Rental extension has already requested	Variable	Storage
Customer notified	Variable	Storage
Customer should pay insurance for each car rentals	Variable	Storage
Contact car Rentals Company in urgent cases	Variable	Storage
Customer should not leave the car when the car damaged until the car rentals receive it	Variable	Storage
Customer could service the car in Urgent cases.	Variable	Storage
Customer could authorize another driver for a car in argent cases.	Variable	Storage
The renter must have a valid driver's license	Variable	Maintain
The car must be insured	Variable	Maintain
The renter should not visit the countries that the insurance not covered	Variable	Storage
The car should be driven by renter only	Variable	Storage
The driver must be over 25	Variable	Maintain
The contract is signed	Variable	Storage
The car is received	Variable	Storage

Table 6.4 Customer Agent Beliefs

6.4.1.3.2 Beliefs of Car Rental Clerk Agent

A notification must be made about the beliefs that are captured from obligations, permissions, and constraints of the role. These beliefs are considered as initial beliefs for both agents (customer and car rental clerk). Therefore, they do not belong to any goals or plans.

Belief	Type	Purpose
Agent Id	Constant	Storage
Customer requested reservation	Variable	Storage
Customer passed car rental regulation	Variable	Storage
Customer does not pass car rental regulation	Variable	Storage
Customer demands are available	Variable	Storage
Customer demands are not available and reservation rejected	Variable	Storage
Car received by customer	Variable	Storage
Customer used loyalty points for payment	Variable	Storage
Rental fee is paid	Variable	Storage
Rental fee is not paid	Variable	Storage
Rental transaction closed	Variable	Storage
Case transferred to court	Variable	Storage
Customer requested rental extension	Variable	Storage
Rental extension confirmed	Variable	Storage
Rental extension rejected	Variable	Storage
Service plan required	Variable	Storage
Service date allocated for car	Variable	Storage
Service is performed and car is ready	Variable	Storage
The rentier should satisfy customers' demands.	Variable	Achieve
The rentier should apply car rental rules.	Variable	Maintain
The rentier must be keep cars papers and documents valid.	Variable	Maintain
A car should be insured before given to renter.	Variable	Maintain
Local taxes must be collected on the rental charge.	Variable	Maintain
The rental would not exceed the mileage more than 10% over the normal mileage for the service	Variable	Maintain
A car scheduled for service may be used	Variable	Storage
A car may have to be rented from a competitor	Variable	Storage
Rented cars must meet local legal requirements	Variable	Maintain

Table 6.5 Car Rental Clerk Agent Beliefs

6.4.1.4 Goals Model

In this model, we describe the goals of the customer agent. The following section show how the customer agent goals are identified.

6.4.1.4.1 Identifying Agent Goals

In this section, we describe how to obtain the goals of the customer agent through the renter role. In order to identify the goals of the customer agent and car rental agent, we have to convert all responsibilities for every role into specific goals respectively. In addition, we have to convert the activities for each goal into plans. In the following example, the goals of the customer agent and car rental agent are identified. This is followed by defining the plans for each goal to be achieved.

6.4.1.4.2 Goals for Customer Agent

Goal	Priority	Preconditions	Postconditions	Plans
Request reservation	High	<ul style="list-style-type: none"> • Customer wants to rent a car 	<ul style="list-style-type: none"> • Reservation confirmed. • Reservation rejected. 	<ul style="list-style-type: none"> • <u>Reserve car by phone call</u> • <u>Reserve car by e-mail</u> • <u>Reserve car online</u>
Cancel reservation request	Normal	<ul style="list-style-type: none"> • Customer wants to cancel reservation 	<ul style="list-style-type: none"> • Cancellation confirmed 	<ul style="list-style-type: none"> • <u>Cancel reservation by phone call.</u> • <u>Cancel reservation by Email.</u> • <u>Cancel reservation online</u>
Pay the rental	Above normal	<ul style="list-style-type: none"> • Payment requested by car rental company 	<ul style="list-style-type: none"> • Fee paid • Fee not paid 	<ul style="list-style-type: none"> • <u>Pay rentals by cash</u> • <u>Pay rentals by credit card</u> • <u>Pay rentals by loyalty club points</u>
Pay extra charge	High	<ul style="list-style-type: none"> • Car is damaged • Car returned late 	<ul style="list-style-type: none"> • Fee paid • Fee not paid and transaction transferred to judgment 	<ul style="list-style-type: none"> • <u>Pay for damage costs by cash</u> • <u>Pay for damage costs by credit card</u> • <u>Pay for late return by cash</u> • <u>Pay for late return by credit card</u>
Notify real customer	Normal	<ul style="list-style-type: none"> • Real customer must be notified 	<ul style="list-style-type: none"> • Customer notified 	<ul style="list-style-type: none"> • <u>Notify customer for picking up a car</u> • <u>Remind customer about return date</u> • <u>Notify customer for canceled reservations</u> • <u>Notify customer for rejected reservations</u> • <u>Notify customer for confirmed reservations</u>
Extend the Rental	Normal	<ul style="list-style-type: none"> • Customer decided to extend rental 	<ul style="list-style-type: none"> • Extension rentals confirmed • Extension rentals rejected 	<ul style="list-style-type: none"> • <u>Request to extend rentals</u> • <u>Reply to car rentals proposals</u> • <u>Notify customer extension rejected</u> • <u>Notify customer extension confirmed</u>

Table 6.6 Goals for customer agent

After identifying the goals of the customer agent and car rental agent in the previous step, describing the plans that the roles will use in order to achieve the agents' goals in the car rental system follows. Each agent has a set of goals and each goal may have several plans. Since the number of these plans will be large, we will describe only one goal and its plans as an example to illustrate how the plans are constructed. Due to the fact that there are a large number of plans for each agent, we will introduce in the following example only plans for one goal of each agent in the

system. We have chosen to develop plans that belong to the request reservation goal for the customer agent as well as plans that belong to the make reservation goal for the car rental clerk agent.

6.4.1.4.2.1 Plans for Request Reservation Goal

In this section, plans for the request reservation goal are constructed. Activity diagrams are used as an effective technique to represent such plans. The first plan of the request reservation goal is *reserve by phone call*. Table 6.7 illustrates the *reserve by phone call* plan and the tasks that should be performed in this plan. Each action of the activity diagram represents a task in the plan.

Plan-name:	<i>Reserve by phone call</i>
Preconditions:	<i>Customer decided to reserve by phone</i>
Postconditions:	<i>Reservation confirmed Reservation rejected</i>
Successful internal actions:	<i>Inform the real customer to pickup the car</i>
Failed internal actions:	<i>Try with another car rental company</i>
Plan body	<pre> graph TD subgraph Customer_agent [Customer agent] Start(()) --> Find[Find phone no. of car Rental company] Find --> Dial[Dial] Dial --> Connected{ } Connected -- "[connected]" --> Forward[Forward call] Forward --> Verify{ } Verify -- "[Busy]" --> Dial Verify -- "[No answer]" --> Hangup[Hangup] Hangup --> End(()) end subgraph Real_user [Real user] Request[Request for reservation] Accept[Accept terms] Store[Store reservation data] end Forward --> Request Request --> Accept Accept --> Store Store --> Connected </pre>

Table 6. 7 Reserve by Phone Call Plan

Table 6.8 illustrates the *reserve by e-mail* plan, which explains how the customer agent requests a reservation by sending an e-mail.

Plan-name:	<i>Reserve by e-mail</i>
Preconditions:	<i>Customer decide to reserve e-mail</i>
Postconditions:	<i>Reservation confirmed Reservation rejected</i>
Successful internal actions:	<i>Inform the real customer to pickup the car</i>
Failed internal actions:	<i>Try with another car rental company</i>

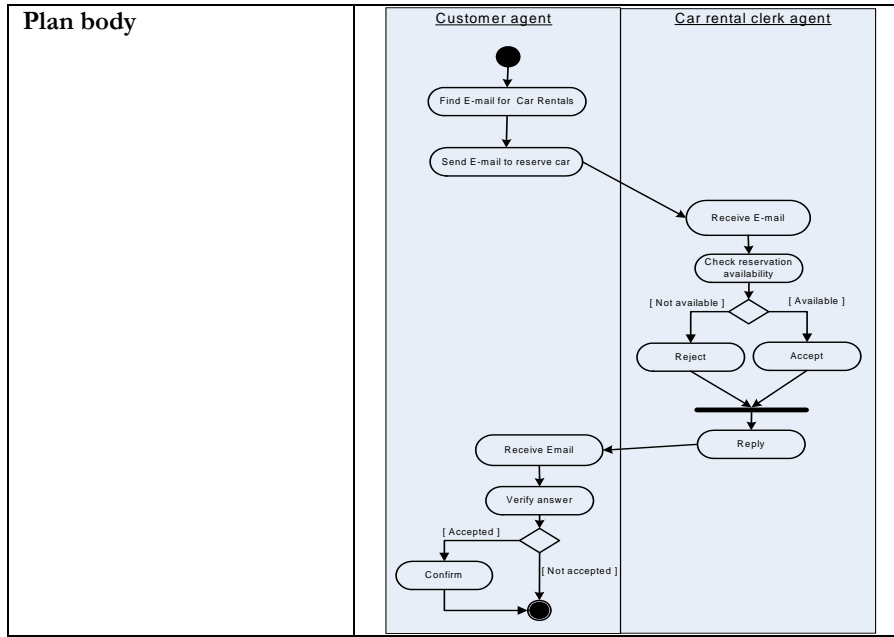


Table 6.8 Reserve by E-mail Plan

Table 6.9 illustrates the *reserve car online* plan, which explains how the customer agent requests a reservation on line.

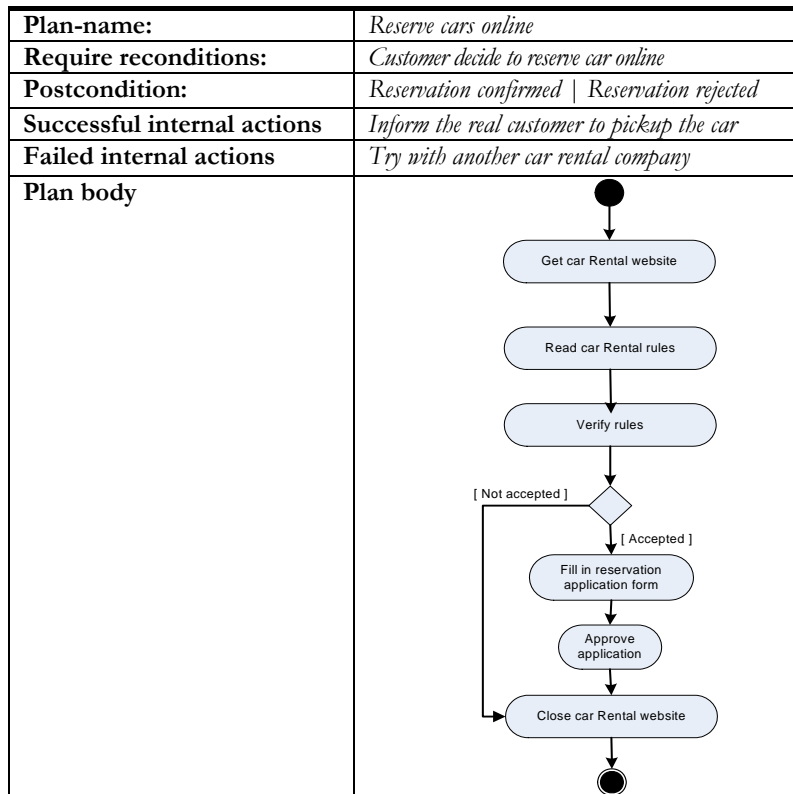


Table 6.9 Reserve Car Online Plan

6.4.1.4.3 Goals for Car Rental Clerk Agent

Goal	Priority	Preconditions	Postconditions	Plans
Make reservation	High	<ul style="list-style-type: none"> Reservation requested by customer 	<ul style="list-style-type: none"> Reservation request accepted Reservation request rejected 	<ul style="list-style-type: none"> <u>Request information.</u> <u>Verify car rental regulations</u> <u>Check Customer's demands.</u>
Cancel reservation	High	<ul style="list-style-type: none"> Reservation cancelled by customer 	<ul style="list-style-type: none"> Reservation cancelled 	<ul style="list-style-type: none"> <u>Cancel reservation</u>
Allocate cars to customers	Normal	<ul style="list-style-type: none"> Reservation request accepted 	<ul style="list-style-type: none"> Car allocated to customer. 	<ul style="list-style-type: none"> <u>Allocate a car to a customer.</u> <u>Allocate a car to a customer from another branch.</u> <u>Allocate a car to a customer from another car rentals company.</u>
Reply customer requests	Normal	<ul style="list-style-type: none"> Customer requested service 	<ul style="list-style-type: none"> Customer pleased 	<ul style="list-style-type: none"> <u>Reply to a customer's reservation request.</u> <u>Reply to a customer's cancellation request.</u> <u>Negotiate with a customer</u>
Handle rental transaction	High	<ul style="list-style-type: none"> Reservation existed 	<ul style="list-style-type: none"> Transaction terminated Transaction transferred to court 	<ul style="list-style-type: none"> <u>Prepare the rentals contract.</u> <u>Obtain insurance for a car to be rented.</u> <u>Confirm payment.</u> <u>Terminate a rental transaction.</u>
Handle extension	Normal	<ul style="list-style-type: none"> Rental extension requested by customer 	<ul style="list-style-type: none"> Rental extension confirmed Rental extension rejected 	<ul style="list-style-type: none"> <u>Manage a rentals extension.</u>
Handle cars service	Normal	<ul style="list-style-type: none"> Cars requires service 	<ul style="list-style-type: none"> Cars are serviced 	<ul style="list-style-type: none"> <u>Book an appointment for a car service.</u> <u>Receive a car service date.</u>

Table 6.10 Goals for Car Rental Agent

6.4.1.4.3.1 Plans for Make Reservation Goal

In this section, the plans of the *make reservation* goal for the car rental clerk agent are constructed. The first plan of the make reservation goal is *request information*. Table 6.11 illustrates the *request information* plan and the tasks that should be performed in this plan. Each activity of the activity diagram represents a task in the plan.

Plan-name:	<i>Request information</i>
Preconditions:	<i>Reservation requested by Customer</i>
Postconditions:	<i>Customer information provided</i>
Successful internal actions:	<i>Verify car rental regulations</i>
Failed internal actions:	<i>Reject reservation request</i>

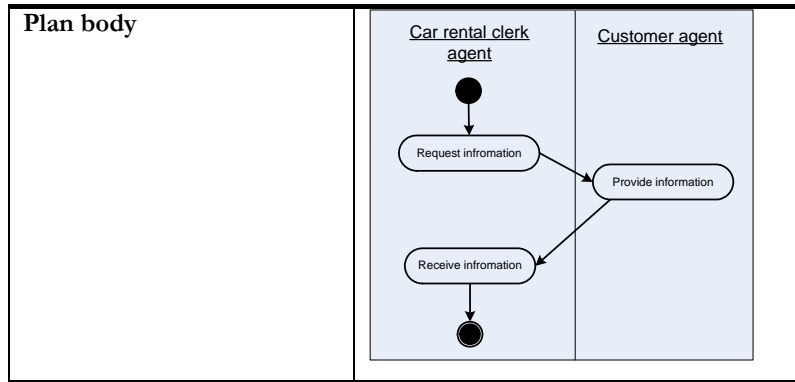


Table 6.11 Request Information Plan

Table 6.12 illustrates the *verify car rental regulations* plan, which explains how the car rental clerk agent verifies whether the customer passed car rental regulations, or not.

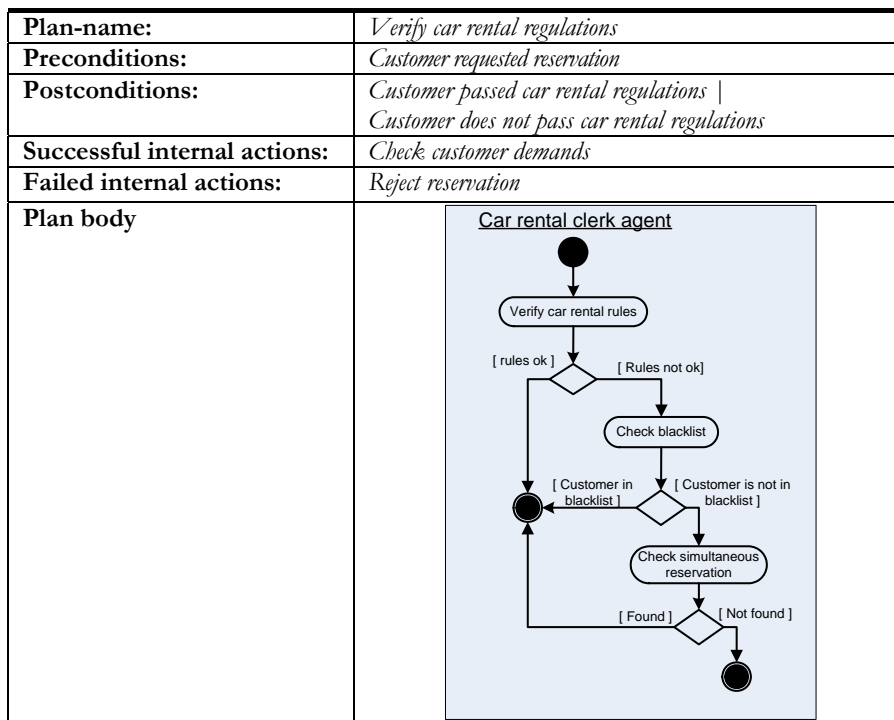


Table 6.12 Verify Car Rental Regulations Plan

Table 6.13 illustrates the *Check customer demands* plan, which explains how the car rental clerk agent checks whether the customer demands are available, or not.

Plan-name:	<i>Check customer demands</i>
Require reconditions:	<i>Customer passed car rental regulations</i>
Postcondition:	<i>Reservation confirmed Reservation rejected</i>
Successful internal actions	<i>Confirm reservation</i>
Failed internal actions	<i>Reject reservation</i>

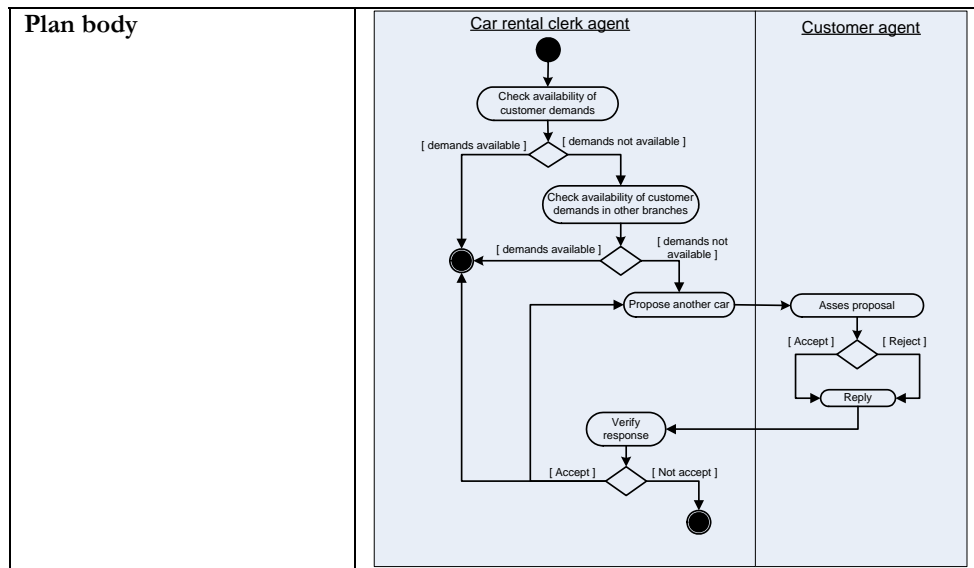


Table 6.13 Check Customer Demands Plan

6.4.1.5 Triggers model

In this model, we describe the triggers of the customer agent as well as car rental clerk agent. The following section shows how agent triggers are captured.

6.4.1.5.1 Triggers Model of Customer Agent

Through this model, all triggers for the customer agent in the system will be identified. Moreover, this model determines the customer agent’s goals and plans that each trigger affects.

Looking at the beliefs model of a particular customer agent and the reservation scenario in UCMs that was built during the system requirements phase, it was found that the scenario begins with precondition (belief) such as “*a customer wants to rent a car*”. In fact, it is possible for this belief to be a true or false belief. If this belief becomes true, this means the real customer wants to rent a car; and this means that the beliefs of the agent have changed. This consequently means that the agent will react based upon the change in its beliefs. This is considered a trigger that motivates the agent to perform a certain action as a reaction such as *request reservation* from the car rental clerk agent. The following table shows a list of the triggers that might occur in all scenarios between the customer agent and car rental clerk agent during system runtime.

The postcondition *reservation canceled* is not considered as a trigger because it was not a precondition for another scenario. It was therefore ignored. However, looking at to the postcondition *request rejected and customer informed*, it is found that it is considered as a trigger because it was a precondition to another scenario, which is for this agent to try to request another reservation from another car rental company. The table 6.14 presents the agent triggers model for the systems' agents.

Trigger-name	Trigger type	Trigger activator	Actions by beneficiary agent
Customer wants to rent a car	Change of belief	Real customer	• Request reservation (Goal).
Customer decides to reserve by a phone	Change of belief	Real customer	• Reserve by a phone (plan).
Customer decides to reserve by an e-mail	Change of belief	Real customer	• Reserve by an e-mail (plan).
Customer decides to reserve online	Change of belief	Real customer	• Reserve Online (plan).
Reservation confirmed	Event	Car rental clerk agent	• Notify real customer to pickup the car (plan).
Reservation rejected	Event	Car rental clerk agent	• Notify real customer about a rejected reservation (plan).
Reservation cancelled	Event	Car rental clerk agent	• Notify real customer about a cancelled reservation (plan).
Customer wants to cancel a reservation	Change of belief	Real customer	• Cancel a reservation request (Goal).
Customer wants to cancel a reservation by a phone	Change of belief	Real customer	• Cancel a reservation by phone (plan).
Customer wants to cancel a reservation by an e-mail	Change of belief	Real customer	• Cancel a reservation by an e-mail (plan).
Customer wants to cancel a reservation online	Change of belief	Real customer	• Cancel a reservation Online (plan).
Cancellation confirmed	Change of belief	Car rental clerk agent	• Inform a real customer (plan).
Payment requested by the Car Rentals.	Change of belief	Car rental clerk agent	• Pay the rental (Goal).
Customer decides to pay rentals in cash	Change of belief	Car rental clerk agent	• Pay rentals in cash (plan).
Customer decides to pay rentals by a credit card.	Change of belief	Car rental clerk agent	• Pay rentals by a credit card (plan).
Customer decides to pay rentals by loyalty club points	Change of belief	Car rental clerk agent	• Pay rentals by loyalty club points (plan).
Car is damaged	Event	Real Customer	• Inform the Car Rental Company (plan).
Real customer should be notified	Change of belief	Car rental clerk agent	• Notify real customer (goal).
Reservation is done and car is allocated to customer	Change of belief	Car rental clerk agent	• Notify real customer to pickup a car (plan).
Car is received by real customer	Event	Real customer	• Remind customer about return date (plan).
Customer decided to extend rentals	Change of belief	Real customer	• Extend the rental (goal). • Request to extend rentals (plan).

Table 6.14 Triggers of Customer Agent

By the same situation, the triggers for the car rental clerk agent are identified. Table 6.15 illustrates the triggers for the car rental clerk agent.

Trigger name	Trigger type	Trigger activator	Actions by beneficiary agent
Customer requested reservation	Change of belief	Customer agent	• Make reservation (goal)
Customer passed car rental regulation	Change of belief	Customer agent	• Check Customer's demands (plan)

Customer does not pass car rental regulation	Change of belief	Customer agent	<ul style="list-style-type: none"> • Reject reservation (plan)
Customer demands are available	Change of belief	Customer agent	<ul style="list-style-type: none"> • Confirm reservation (plan)
Customer demands are not available	Change of belief	Customer agent	<ul style="list-style-type: none"> • Reject reservation (plan)
Rental fee is paid	Change of belief	Customer agent	<ul style="list-style-type: none"> • Achieve a rental transaction (goal).
Rental fee is not paid	Change of belief	Customer agent	<ul style="list-style-type: none"> • Cancel a reservation (plan).
Cancel reservation is requested by customer	Event	Customer agent	<ul style="list-style-type: none"> • Cancel a reservation (goal).
Rental extension requested	Event	Customer agent	<ul style="list-style-type: none"> • Handle extension (goal) • Manage an extension (plan).
Rental extension rejected	Change of belief	Car rental clerk agent	<ul style="list-style-type: none"> • Notify a customer agent (plan) • Propose another car (plan).
Rental extension confirmed	Change of belief	Car rental clerk agent	<ul style="list-style-type: none"> • Notify a customer agent (plan). • Update a car rental database (plan)

Table 6.15 Triggers of the Car Rental Clerk Agent

6.4.2 MAS Architecture Stage

In the agent architecture stage, we described the internal structure (roles, beliefs, goals, plans and triggers) of agents in the system. The next stage of the analysis phase is the MAS architecture stage. The MAS architecture stage describes how the whole multi-agent system is concerned. The MAS architecture stage will perform the following tasks:

- 1) Identify the interactions between agents in the car rental system through the agent interaction model.
- 2) Identify the relationships between agents in the car rental system through the agent relationship model.
- 3) Identify the services that each agent should perform in the system through the agent services model.

6.4.2.1 Agents Interaction Model

This model describes all the interactions that take place between the customer agent and the car rental clerk agent. Figure 5.17 in chapter described the interaction between customer agent and car rental clerk agent and how the interactions diagram are derived from UCMs. The customer requests the agent to reserve a particular car. This request is sent to the car rental clerk agent which requests more detailed information about the customer. The customer agent replies to the car rental clerk agent; who checks the rules of the car rental company and then replies to the customer agent by either acceptance or rejection.

By the same situation, the interactions diagrams for the other UCM scenarios are identified. Figure 6.18 shows the interactions for cancel reservation, car pickup, and rental extension scenarios.

The figure 6.18 describes the interaction diagrams for all UCM scenarios that take place between the customer agent and the car rental clerk agent in the same figure. The reason is to prevent the interference that could happen between the paths in more than one scenario at the same UCM. Therefore, it is preferred to draw each

scenario separately as is in the figure 6.18. The previous diagram shows all interactions between the customer agent and the car rental clerk agent.

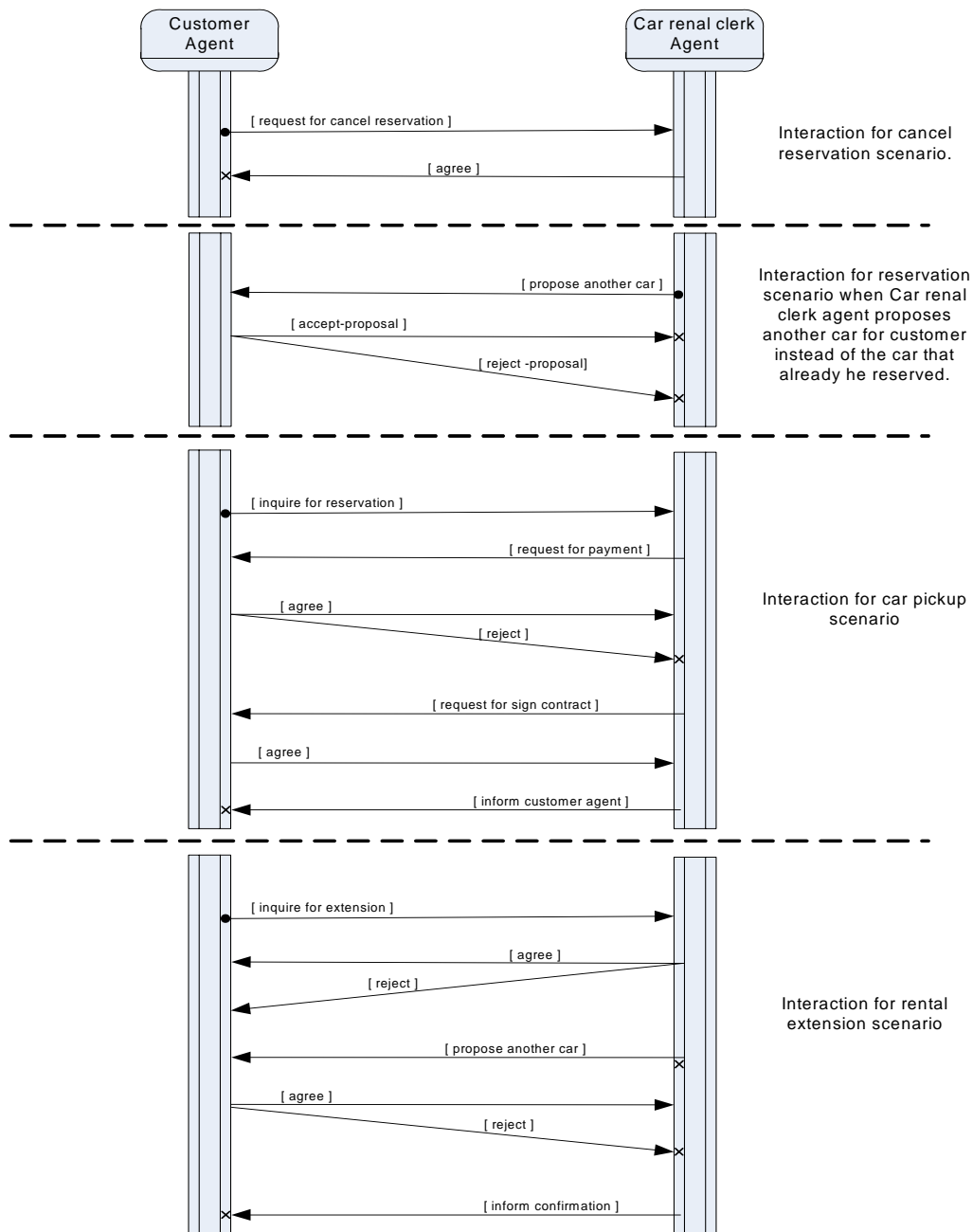


Figure 6.18 Interaction Diagrams between Customer Agent and Car Rental Clerk Agent

6.4.2.2 Agents Relationships Model

This model describes the relationships between the customer agent and the car rental agent. These relationships are represented in a dependency diagram. A dependency diagram is captured from UCMs. Each path's segment connecting two agents generates a dependency in the dependency diagram. Figure 6.19 shows dependencies between the customer agent and the car rental clerk agent. Firstly, the

customer agent’s dependencies are stated, then the car rental clerk agent’s dependencies. The customer agent depends on the car rental clerk agent of the car rental company to handle reservation requests. This dependency is classified as “goal dependency” because the customer agent depends on the car rental clerk agent to achieve a specific goal. This goal is called *requesting reservation*. It also depends on the car rental clerk agent to achieve the canceling reservation goal when the customer wishes to cancel the reservation, or to provide him/her with his/her list reservation information, or to extend his/her rental.

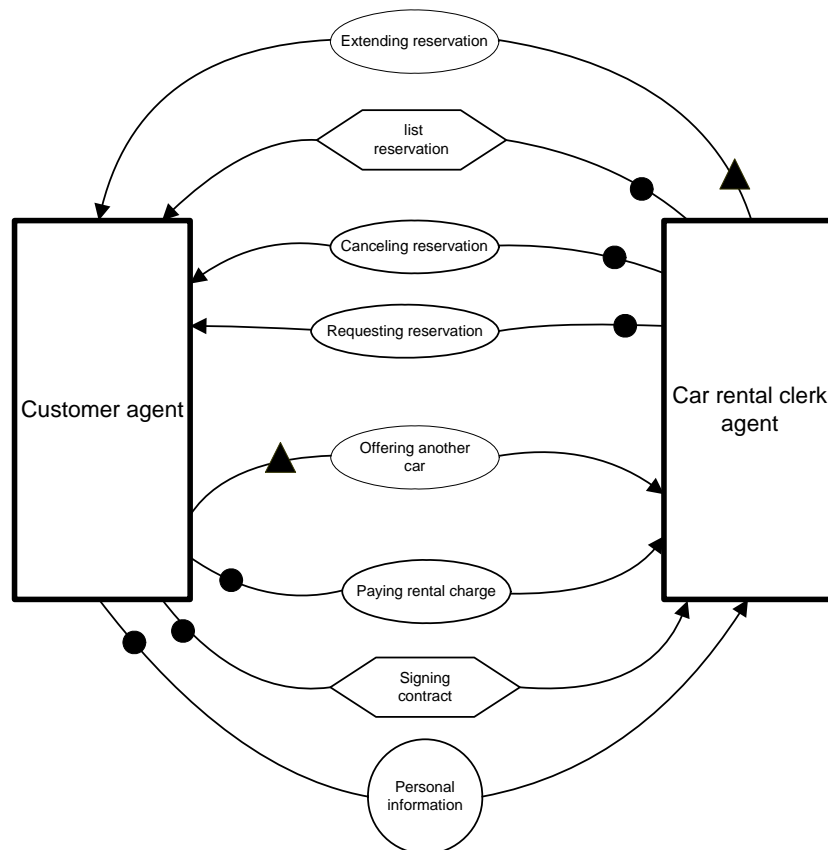


Figure 6.19 Dependency Diagram between Customer Agent and Car Rental Clerk Agent

6.4.2.3 Agents Services Model

This model is used as a directory that assists the system's agents to recognize what services are offered by each agent in the system. In addition, it provides more detailed information about such services, such as service description, service expiry date, time of availability, and cost. The agent services model is derived from the use case diagrams that were developed in the system scenario model. Each use case could be considered as a service that the agent offers. Table 6.16 illustrates the agent services model for the car rental clerk agent of the car rental company.

Agent	Service	Expire Date	Time	Cost
Car rental clerk agent	Reply to a customer inquiries	Open	None-stop	Free
Car rental clerk agent	Handel a reservation request	Open	8:00 am to 4:00 pm	Free
Car rental clerk agent	Handle rental	Open	8:00 am to 4:00 pm	Free
Car rental clerk agent	Handel a car service	Open	8:00 am to 2:00 pm	Free

Table 6.16 Agent Services Model

6.5 Design Phase

The design phase consists of three tasks. The first task is to construct the agent container for each agent in the system. The agent container represents agent behaviour, which can be modularized, and decomposed into roles specifications that are used by agents. The core part of the agent specification is to define beliefs, goals, plans, capabilities and triggers of the agent and place them in the appropriate agent part.

The second task is building the agent inter-communication model by means of FIPA-ACL protocols. These protocols describe the conversations between the customer agent and the car rental clerk agent in more details than the developed interaction model in the analysis phase.

The final task is designing the Directory Facilitator (DF) model for the car rental clerk agent services model that was developed in the analysis phase. The DF model serves as the “yellow pages” directory for the system agents.

6.5.1 Customer Agent Container

The customer agent container contains all the important characteristics needed by the agent to start working. It includes all the models developed in the analysis phase that are related to customer agent behaviors. The customer agent container is divided into different components where each are represented in a certain model. Each model and its programming aspects will be defined in order to fit the agent platform (Jadex platform as example).

6.5.1.1 Customer Agent Beliefs

The first part of the customer agent container is the customer agent beliefs. The customer beliefs model is revised and modified in order to fit the design specifications. Some important details were added during the design stage. A new field “class” was added to contain *Integer*, *String* or *Boolean*. The initial value for that field depends on the type of belief. In addition, the “category” field was added which refers to “f” for the beliefs that store exactly one fact; or refers to “s” for the belief that stores a set of facts. Table 6.17 provides a detailed description of all the additions of a customer agent that was setup during the analysis stage.

Belief	Type	Purpose	Class	Initial Value	Cat.
Agent Id	Constant	Storage	String	Customer	F
Customer wants to rent a car	Variable	Storage	Boolean	True	F
Customer decided to reserve by phone	Variable	Storage	Boolean	True	F
Customer decided to reserve by e-mail	Variable	Storage	Boolean	False	F
Customer decided to reserve car online	Variable	Storage	Boolean	False	F
Reservation confirmed	Variable	Storage	Boolean	False	F
Reservation rejected	Variable	Storage	Boolean	False	F
Customer wants to cancel reservation	Variable	Storage	Boolean	False	F
Customer decided to cancel reservation by phone	Variable	Storage	Boolean	False	F
Customer decided to cancel reservation by e-mail	Variable	Storage	Boolean	False	F
Customer decided to cancel reservation online	Variable	Storage	Boolean	False	F
Cancellation confirmed	Variable	Storage	Boolean	False	F
Payment requested by car Rentals Company	Variable	Storage	Boolean	False	S
Customer decided to pay rental cash	Variable	Storage	Boolean	False	F
Customer decided to pay rental by credit card	Variable	Storage	Boolean	False	F
Customer decided to pay rental by loyalty club points	Variable	Storage	Boolean	False	F
Rental fee paid	Variable	Storage	Boolean	False	F
Rental fee not paid	Variable	Storage	Boolean	False	F
Car is damaged	Variable	Storage	Boolean	False	F
Car returned late	Variable	Storage	Boolean	False	F
Customer decided to pay extra charge cash.	Variable	Storage	Boolean	False	F
Customer decided to pay extra charge by credit card	Variable	Storage	Boolean	False	F
Fee paid	Variable	Storage	Boolean	False	F
Fee not paid and transaction transferred to court	Variable	Storage	Boolean	False	F
Real customer must be notified	Variable	Achieve	Boolean	False	F
Reservation already done and car allocated to the customer	Variable	Storage	Boolean	False	F
The car received by customer	Variable	Storage	Boolean	False	F
Cancel reservation is already requested by customer	Variable	Storage	Boolean	False	F
Reservation Request is already done by customer	Variable	Storage	Boolean	False	F
Customer notified	Variable	Storage	Boolean	False	F
Customer confirmed	Variable	Storage	Boolean	False	F
Customer decided to extend rentals	Variable	Storage	Boolean	False	F
Another offer proposed by car rental clerk agent	Variable	Storage	Boolean	False	F
Rental extension has already requested	Variable	Storage	Boolean	False	S
Customer notified	Variable	Storage	Boolean	False	F
Customer should pay insurance for each car rentals	Variable	Storage	Boolean	True	F
Contact car Rentals Company in urgent cases	Variable	Storage	Boolean	True	F
Customer should not leave the car when the car damaged until the car rentals receive it	Variable	Storage	Boolean	True	F
Customer could service the car in Urgent cases.	Variable	Storage	Boolean	True	F
Customer could authorize another driver for a car in argent cases.	Variable	Storage	Boolean	True	F
The renter must have a valid driver's license	Variable	Maintain	Boolean	True	F

The car must be insured	Variable	Maintain	Boolean	True	F
The renter should not visit the countries that the insurance not covered	Variable	Storage	Boolean	True	F
The car should be driven by renter only	Variable	Storage	Boolean	True	F
The driver must be over 25	Variable	Maintain	Boolean	True	F
The contract is signed	Variable	Storage	Boolean	True	F
The car is received	Variable	Storage	Boolean	True	F

Table 6.17 Beliefs of a Customer Agent

6.5.1.2 Customer Agent Goals

The goals of the customer agent that was developed during the analysis stage are revised and modified in order to fit the design specification. Some important details were added during the design stage. A new field “type” was added to classify the agent goals that were developed in the analysis phase according to the proposed types from Jadex. In order to classify these goals a *type* field is added to the goal model as shown in table 6.18.

Goal	Type	Priority	Preconditions	Postconditions	Plans
Request reservation	Achieve goal	High	<ul style="list-style-type: none"> • Customer wants to rent a car 	<ul style="list-style-type: none"> • Reservation confirmed. • Reservation rejected. 	<ul style="list-style-type: none"> • <u>Reserve car by phone call</u> • <u>Reserve car by e-mail</u> • <u>Reserve car online</u>
Cancel reservation request	Achieve goal	Normal	<ul style="list-style-type: none"> • Customer wants to cancel reservation 	<ul style="list-style-type: none"> • Cancellation confirmed 	<ul style="list-style-type: none"> • <u>Cancel reservation by phone call.</u> • <u>Cancel reservation by Email.</u> • <u>Cancel reservation online</u>
Notify real customer	Perform goal	High	<ul style="list-style-type: none"> • Real customer must be notified 	<ul style="list-style-type: none"> • Customer notified 	<ul style="list-style-type: none"> • <u>Notify customer for picking up a car</u> • <u>Remind customer about return date</u> • <u>Notify customer for cancelled reservations</u> • <u>Notify customer for rejected reservations</u> • <u>Notify customer for confirmed reservations</u>
Pay the rental	Achieve goal	Above normal	<ul style="list-style-type: none"> • Payment requested by car rental company 	<ul style="list-style-type: none"> • Fee paid • Fee not paid 	<ul style="list-style-type: none"> • <u>Pay rentals by cash</u> • <u>Pay rentals by credit card</u> • <u>Pay rentals by loyalty club points</u>

Pay extra charge	Achieve goal	High	<ul style="list-style-type: none"> • Car is damaged • Car returned late 	<ul style="list-style-type: none"> • Fee paid • Fee not paid and transaction transferred to judgment 	<ul style="list-style-type: none"> • <u>Pay for damage costs by cash</u> • <u>Pay for damage costs by credit card</u> • <u>Pay for late return by cash</u> • <u>Pay for late return by credit card</u>
Extend the Rental	Achieve goal	Normal	<ul style="list-style-type: none"> • Customer decided to extend rental 	<ul style="list-style-type: none"> • Extension rentals confirmed • Extension rentals rejected 	<ul style="list-style-type: none"> • <u>Request to extend rentals</u> • <u>Reply to car rentals proposals</u> • <u>Notify customer extension rejected</u> • <u>Notify customer extension confirmed</u>

Table 6.18 Revised Customer Agent Goals

6.5.1.3 Plans Request Reservation Goal

In this section, the plans, which have been developed during the analysis phase, are revised by adding a new field called *type*. This field is added to fit the plan structure of the Jadex framework. The following tables illustrate the plans of the request reservation goal for the customer agent with the new field type.

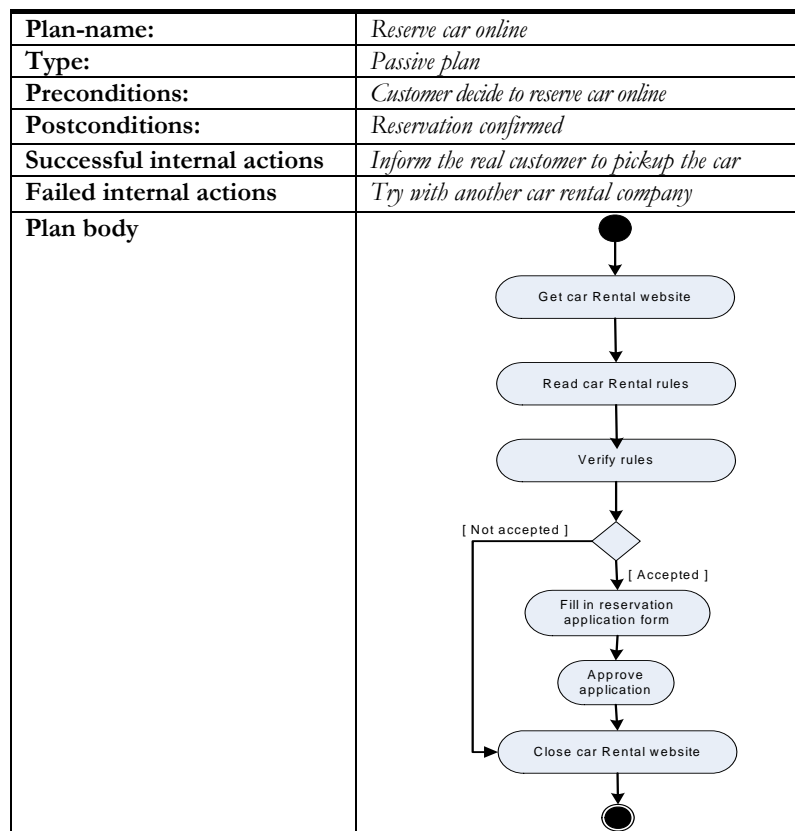


Table 6.19 Revised Reserve Car Online Plan

By the same situations, all the plans that were developed in the analysis phase were revised by adding the *type* field. Table 6.20 shows the type of each plan.

Goal	Plan	Type:
<i>Request reservation</i>	<i>Reserve cars online</i>	<i>Passive plan</i>
<i>Request reservation</i>	<i>Reserve by phone call</i>	<i>Passive plan</i>
<i>Request reservation</i>	<i>Reserve by e-mail</i>	<i>Passive plan</i>
<i>Make reservation</i>	<i>Request information</i>	<i>Passive plan</i>
<i>Make reservation</i>	<i>Verify car rental regulations</i>	<i>Passive plan</i>
<i>Make reservation</i>	<i>Check customer demands</i>	<i>Passive plan</i>

Table 6.20 Plan Types

6.5.1.4 Capabilities

The capabilities are a group of goals, beliefs, and plans grouped together in a package in order to be used when they are needed to do a certain task in the system. For example, the customer agent can possess the *reservation* capability, which points out what the customer agent can do to request and cancel the reservation. The following structure shows the details of the reservation capability:

Reservation capability:

Begin // *Reservation capability,*

Beliefs:

*Customer wants to rent a car,
Reservation confirmed,
Reservation rejected.
Customer wants to cancel reservation
Cancellation confirmed.*

Goals:

Begin // *goals*

Request reservation goal

Plans:

*Reserve by phone call,
Reserve by e-mail,
Reserve car online,*

Cancel reservation request goal

Plans:

*Cancel reservation online.
Cancel reservation by phone call,
Cancel reservation by Email,*

End // *goals*

End // *Reservation capability*

In the same way, the car rental clerk agent can possess several capabilities. For example, the car rental clerk agent can possess the *manage reservation* capability which points out what the car rental clerk agent can do to manage the reservation process.

Manage reservation capability:

Begin // *Manage reservation capability,*

Beliefs:

Reservation requested by customer

Customer requested reservation
Customer passed car rental regulation
Customer does not pass car rental regulation
Customer demands are available
Customer demands are not available and reservation rejected
Reservation request accepted
Reservation request rejected
Reservation cancelled by customer
Reservation cancelled

Goals:

Begin // goals

Make reservation goal

Plans:

Request information
Verify car rental regulations
Check Customer's demands

Cancel reservation goal.

Plans:

Cancel reservation.

End // goals

End // Manage reservation capability

6.5.1.5 Triggers

This section describes the triggers that have been developed in the analysis phase handled by the Jadex platform. The triggers that developed in the analysis phase are considered as conditions that trigger plans or goals when some beliefs change or events occur, and beliefs that are stored as expressions and evaluated dynamically on demand.

6.5.2 Inter-Agent Communication Model

In this section of the design phase, the inter-agent communication model is described. It is a detailed model that shows and describes all conversations between agents in the system using a standard agent communication language, such as FIPA Agent Communication Language ACL [FIPA ACL].

Due to the large number of conversation messages between the agents in the system, we will describe the conversations that will take place between the customer agent and the car rental agent only. The interaction diagrams that were developed in the analysis phase are used to describe the conversation messages according to the FIPA-ACL protocol patterns. This model is derived by transforming the interaction diagrams between the customer agent and car rental agent into conversation messages according to FIPA protocol patterns. Figure 6.20 illustrates how the interaction diagram is transferred into a FIPA request protocol.

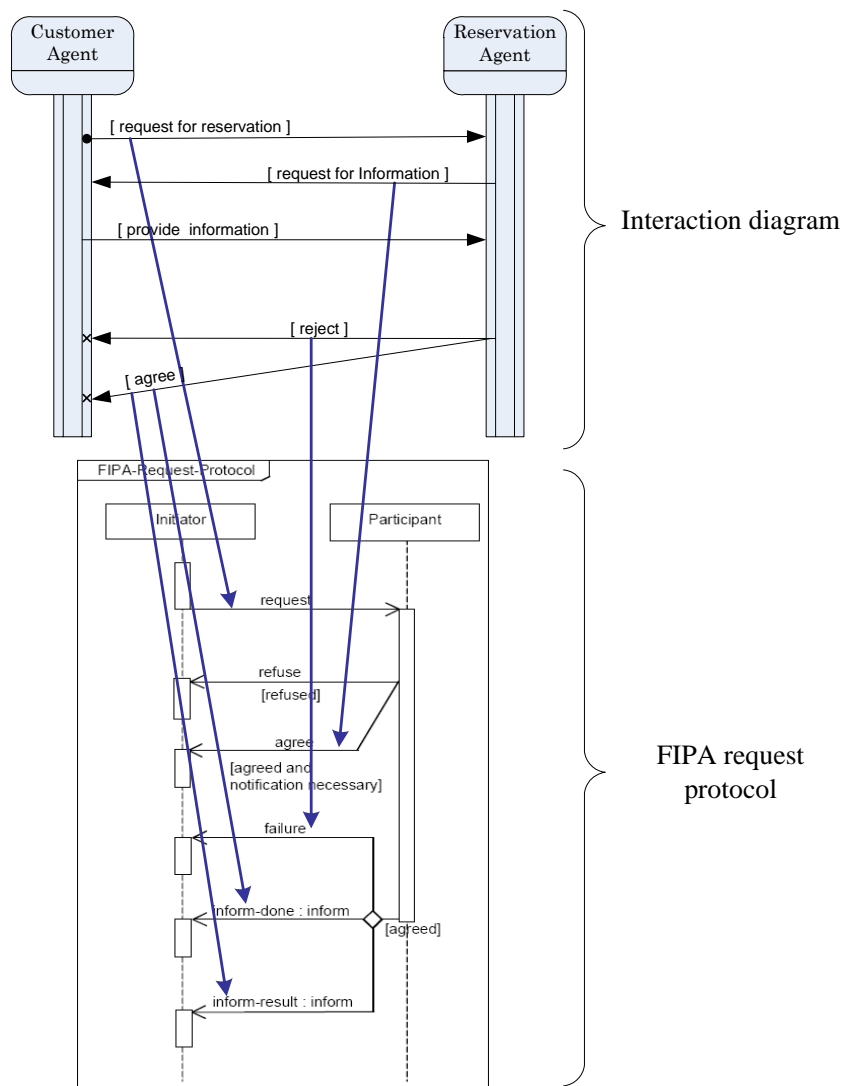


Figure 6.20 The Correspondence Between Interaction Diagrams and FIPA Protocols

The following examples describe the message's forms, which will be sent from the customer agent to the car rental clerk agent and visa versa.

- i. Customer agent requests the car rental clerk agent to reserve a car.


```
(request
  :sender (agent-identifier :Customer_agent)
  :receiver (agent-identifier :Car_rental_clerk_agent)
  :content
    "Reserve group B car for rent"
  :reply-with reserve-car
  :language fipa-sl
  :ontology Car_rental
  :protocol fipa-request interaction)
```
- ii. Car rental clerk agent answers that it agrees to the request but it requests extra information.


```
(agree
  :sender (agent-identifier :Car_rental_clerk_agent)
  :receiver (agent-identifier :Customer_agent))
```

```

:content
  ((action (agent-identifier :Car_rental_clerk_agent)
    (agree))
:protocol fipa-agree
:language fipa-sl)

```

- iii. Car rental clerk agent requests the customer agent to provide it with extra information.

```

(request
:sender (agent-identifier :Car_rental_clerk_agent)
:receiver (agent-identifier :Customer_agent)
:content
  "Extra information needed"
:language fipa-sl
:ontology E-rent
:protocol fipa-request interaction)

```

- iv. The customer agent informs the car rental clerk agent that the extra information is provided.

```

(inform
:sender (agent-identifier :Customer_agent)
:receiver (set (agent-identifier :Car_rental_clerk_agent))
:content
  "Extra information provided"
:language fipa-sl)

```

- v. The car rental clerk agent informs the customer agent that the reservation is confirmed.

```

(Confirm
:sender (agent-identifier :Car_rental_clerk_agent)
:receiver (set (agent-identifier :Customer_agent))
:content
  "Reservation confirmed"
:in-reply-to reserve-car
:language fipa-sl)

```

- vi. The car rental clerk agent informs the customer agent that the reservation request is rejected.

```

(refuse
:sender (agent-identifier :Car_rental_clerk_agent)
:receiver (agent-identifier :Customer_agent)
:content
  (reserve-car)
  (customer in blacklist)"
:in-reply-to reserve-car
:language fipa-sl)

```

- vii. Customer agent requests car rental clerk agent to cancel reservation.

```

(request
:sender (agent-identifier :Customer_agent)
:receiver (agent-identifier :Car_rental_clerk_agent)
:content
  "Cancel reservation"
:reply-with reserve-car
:language fipa-sl
:ontology E-rent
:protocol fipa-request interaction)

```

viii. Car rental clerk agent answers, the cancellation request is accepted.

```
(agree
:sender (agent-identifier :Car_rental_clerk_agent)
:receiver (agent-identifier :Customer_agent)
:content
  ((action (agent-identifier :Car_rental_clerk_agent)
  (agree))
:protocol fipa-agree
:language fipa-sl)
```

ix. Agent car rental clerk asks the customer to submit its proposal for selecting another car.

```
(propose
:sender (agent-identifier :Car_rental_clerk_agent)
:receiver (set (agent-identifier :Customer_agent))
:content
  ((action (agent-identifier :Car_rental_clerk_agent)
  (Propose another group B car for rent))
:ontology E-rent
:in-reply-to reserve-car
:language fipa-sl)
```

x. Agent customer informs the car rental clerk that it accepts an offer from the car rental clerk to select another car.

```
(accept-proposal
:sender (agent-identifier :Customer_agent)
:receiver (set (agent-identifier : Car_rental_clerk_agent))
:in-reply-to (propose another group B car for rent)
:content
  ((action (agent-identifier : Car_rental_clerk_agent)
  (propose another group B car for rent))
:language FIPA-SL)
```

xi. Agent customer informs the car rental clerk that it rejects an offer from the car rental clerk agent to select another car.

```
(reject-proposal
:sender (agent-identifier : Customer_agent)
:receiver (set (agent-identifier : Car_rental_clerk_agent))
:content
  ((action (agent-identifier : Car_rental_clerk_agent)
  (propose another group B car for rent))
:in-reply-to (propose another group B car for rent))
```

By the same situation, we can describe all the conversation messages that will take place in the system.

6.5.3 Directory Facilitator Model

The DF is a centralized registry of entries, which associate service descriptions to agents. The DF is used for *adding* an entry or *searching* for services. The DF must be able to perform the following functions: service registration, service deregistration, service modification, and search for services. The DF will be constructed according to the Jadex framework. Jadex considers a Directory Facilitator as an agent that is specified by FIPA Agent Management Specification [FIPA].

6.6 Implementation Phase

The implementation (or construction) phase is the point in the development process when the solution is actually constructed.

6.6.1 Implementing the Case Study with Jadex

In this section, we discuss the implementation process of the case study “Car Rentals Company” using the Jadex framework. In this implementation process, we explain how the design models are handled by Jadex. We then present the implementation code as Jadex proposed. Most of the design models are ready to be employed by Jadex. In the implementation phase, we will describe only how the customer agent is implemented. Due to a matter of time constraints, we will not discuss the car rental clerk agent.

6.6.1.1 Starting an Agent

Due to matter of time and we will not discuss how to set up the Jadex environment because it is out of the scope of this work. Starting up an agent begins with the creating an agent. The agent is created according to the agent container that was developed in the design phase. Each agent container represents an agent definition file in Jadex.

Firstly, the creation a new agent definition file (ADF) called Customer.agent.xml. was established. In this file, all important agent startup properties are defined in a way that complies with the Jadex schema specification. The first attribute of the agent is its type name, which must be the same as the file name (similar to Java class files). In this case, it is set to Customer. Additionally one can specify a package attribute, which has a similar meaning as in Java programs and serves for grouping purposes only (the package name will need to be altered with respect to the actually used directory structure). All plans and other Java classes from the agent's package are automatically known and need not to be imported via an import tag. The following XML code describes the details of the customer ADF.

```
<!-- CustomerAgent -->
<agent xmlns="http://jadex.sourceforge.net/jadex"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://jadex.sourceforge.net/jadex
      http://jadex.sourceforge.net/jadex-0.96.xsd"
      name="Customer"
      package="carrental.app">
</agent>
```

6.6.1.1.1 Defining Beliefs in the ADF

Beliefs stand for the agent's knowledge about its environment and itself. These beliefs can be any Java objects. According to Jadex, beliefs that were constructed during the design phase are classified into to types of beliefs as follows: a belief that represents one fact or a set of beliefs, which represents a set of facts. The customer agent beliefs are stored in a beliefbase. This beliefbase is the container for all the facts known by the customer agent. The customer agent's beliefs are defined in the ADF and are accessed and modified from plans. To define a single valued belief or a multi-valued belief set in the ADF, the developer has to use the corresponding <belief> or

<beliefset> tags and has to provide a name and a class. The name is used to refer to the fact(s) contained in the belief. The class specifies the (super) class of the fact objects that can be stored in the belief. The default fact(s) of a belief may be supplied in enclosed <fact> tags. The following XML code shows an example for different types of beliefs that are represented and defined in ADF.

```

<agent>
  <beliefs>
    <belief name="name" class="Boolean">
      <fact>"John"</fact>
    </belief>
    <belief name="my_location" class="Location">
      <fact>new Location("Hamburg")</fact>
    </belief>
    <beliefset name="my_friends" class="String">
      <fact>"Alex"</fact>
      <fact>"Blandi"</fact>
      <fact>"Charlie"</fact>
    </beliefset>
    <beliefset name="my_opponents" class="String">
      <facts>Database.getOpponents()</facts>
    </beliefset>
  </beliefs>
</agent>

```

The beliefs of the customer agent can be implemented as the following XML code:

```

<!-- CustomerAgent -->
<agent xmlns="http://jadex.sourceforge.net/jadex"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://jadex.sourceforge.net/jadex
    http://jadex.sourceforge.net/jadex-0.96.xsd"
  name="Customer"
  package="jadex.tutorial">
  <beliefs>
    <belief name="agent_Id" class="String">
      <fact>"Customer"</fact>
    </belief>
    <belief name="Customer wants to rent a car" class="Boolean">
      <fact>"True"</fact>
    </belief>
    <belief name="Customer decided to reserve by phone" class="Boolean">
      <fact>"True"</fact>
    </belief>
    <belief name="Customer decided to reserve by e-mail" class="Boolean">
      <fact>"False"</fact>
    </belief>
    <belief name="Customer decided to reserve car online" class="Boolean">
      <fact>"False"</fact>
    </belief>
    <belief name="Reservation confirmed" class="Boolean">
      <fact>"True"</fact>
    </belief>
  </beliefs>
</agent>

```

6.6.1.1.2 Defining Goals in the ADF

Now we discuss how the goals of the customer agent can be constructed step-by-step. The following XML code shows how the customer agent goals are represented by Jadex. Let us start with the most important goal of the customer agent. This goal is classified as an *achieve* goal. The goal is named *request_reservation*. It is created whenever the precondition is satisfied (see creation condition in the following XML code).

```

<achievegoal name="request_reservation">
  <creationcondition>
    $beliefbase.Customer_wants_to_rent_a_car.
  </creationcondition>

```

```

<unique/>
<deliberation>
  <inhibits ref="Cancel reservation request "/>
</deliberation>
</achievegoal>

```

The goal is *<unique/>* meaning that the customer agent will not pursue two goals to reserve car at the same time. Moreover, the *<deliberation>* settings specify that the request reservation goal is more important than the *cancel_reservation_request* goal.

In this section we will discuss another kind of goal: the *perform* goal. In the example of the customer agent the goal *notify_real_customer* is classified as a perform goal. You can see that the perform goal *notify_real_customer* refines some BDI flags (see table 6.21) to achieve the desired behaviour. By allowing the goal to redo activities (*retry="true"*), it is assured that the agent does not conclude to knock off that goal after having performed one notification, but instead notifies the real customer as long as the reservation request is active as described in the context condition. Even when the customer agent only knows of one notification plan, it will reuse this plan and perform the notifications until the goal succeeded.

```

<performgoal name="notify_real_customer" retry="true" exclude="when_succeeded">
  <contextcondition>
    !$beliefbase.reservation_request_rejected & &
    !$beliefbase.reservation_request_canceled
  </contextcondition>
</performgoal>

```

Name	Default	Possible Values
retry	true	{ true, false}
retry	true	{ true, false}
retrydelay	0	positive long value
recur	false	{ true, false} (for maintain goals only)
recurdelay	0	positive long value (for maintain goals only)
exclude	"when_tried"	{ "when_tried", "when_succeeded", "when_failed", "never"}
posttoall	false	{ true, false}
randomselection	false	{ true, false}

Table 6.21 Common Goal Attributes (BDI flags)

In the following XML code, we present the agent definition file ADF with some goals of the customer agent. We are not going to set all the goals; all we need is to show the developer how goals are constructed. The goals are captured from the goals model and their preconditions can be captured from the triggers model.

```

<!-- CustomerAgent -->
<agent xmlns="http://jadex.sourceforge.net/jadex"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://jadex.sourceforge.net/jadex
    http://jadex.sourceforge.net/jadex-0.96.xsd"
  name="Customer"
  package="jadex.tutorial">
  <beliefs>
    <belief name="agent_Id" class="String">
      <fact>"Customer"</fact>

```

```

        </belief>
        <belief name="customer_wants_to_rent_a_car" class="Boolean">
            <fact>"true"</fact>
        </belief>
        <belief name="customer_decided_to_reserve_car_by_phone" class="Boolean">
            <fact>"true"</fact>
        </belief>
        <belief name="customer_decided_to_reserve_car_by_email" class="Boolean">
            <fact>"false"</fact>
        </belief>
        <belief name="customer_decided_to_reserve_car_online" class="Boolean">
            <fact>"false"</fact>
        </belief>
        <belief name=" Reservation confirmed" class="Boolean">
            <fact>"true"</fact>
        </belief>
    </beliefs>

    <goals>
        <achievegoal name="request_reservation">
    <creationcondition>
        $beliefbase.Customer_wants_to_rent_a_car.
    </creationcondition>
    <unique/>
    <deliberation>
        <inhibits ref="cancel_reservation_request "/>
    </deliberation>
        <targetcondition>
            $beliefbase.reservation_confirmed ||
            $beliefbase.reservation_rejected
        </targetcondition>
    </achievegoal>
        <achievegoal name="cancel_reservation_request">
    <creationcondition>
        $beliefbase.customer_wants_to_cancel_reservation.
    </creationcondition>
    <unique/>
    </achievegoal>
        <achievegoal name="pay_rental ">
    <creationcondition>
        $beliefbase. Payment_requested_by_car_cental_company.
    </creationcondition>
    <unique/>
    </achievegoal>
        <performgoal name="notify_real_customer" retry="true" exclude="never">
    <contextcondition>
        !$beliefbase.reservation_request_rejected &amp;&amp;
        !$beliefbase. reservation_request_canceled
    </contextcondition>
    </performgoal>
    </goals>
</agent>

```

6.6.1.1.3 Defining Plans in the ADF

Plans represent actions that the agent can perform. Depending on the current situation, plans are selected in response to occurring events or goals. The selection of plans is done automatically by the system. In Jadex, plans are composed of two parts: A plan head and a corresponding plan body. The plan head is declared in ADF whereas the plan body is realized by a concrete Java class. Therefore, the plan head defines the circumstances under which the plan body is instantiated and executed.

For each plan head, several attributes and contained elements can be defined as in the following XML code. The first attribute that has to be provided is the name of the plan. The second is the priority of a plan, which describes its preference in comparison to other plans. Therefore, it is used to determine which candidate plan will be chosen for a certain event occurrence, favoring higher priority plans (random selection, if activated, applies only to plans of equal priority). Per default, all applicable plans have a default priority of 0 and are selected in order of appearance (or randomly when the corresponding BDI flag is set).

Now we explain how the plans of the customer agent can be constructed in ADF. The following XML code shows how the customer agent plans are represented by Jadex. The developer should classify the plans into either service or passive. Therefore, in this example we consider the plans as being passive plans. Let us start with the plans that are related with the goal request reservation of the customer agent. These plans are (reserve car by phone call, reserve car by e-mail, and reserve car online). The plans are captured from the goals and the triggering conditions are captured from the triggers model.

```

<plans>
  <plan name="reserve_car_by_phone_call">
    <body> new reserve_car_by_phone_callPlan() </body>
    <trigger>
      <condition>
        $beliefbase.customer_decided_to_reserve_car_by_phone
      </condition>
    </trigger>
    <contextcondition>
      $beliefbase.reservation_confirmed || $beliefbase.reservation_rejected
    </contextcondition>
  </plan>
  <plan name="reserve_car_by_email">
    <body> new reserve_car_by_emailPlan() </body>
    <trigger>
      <condition>
        $beliefbase.customer_decided_to_reserve_car_by_email
      </condition>
    </trigger>
    <contextcondition>
      $beliefbase.reservation_confirmed || $beliefbase.reservation_rejected
    </contextcondition>
  </plan>
  <plan name="reserve_car_online">
    <body> new reserve_car_onlinePlan() </body>
    <trigger>
      <condition>
        $beliefbase.customer_decided_to_reserve_car_online
      </condition>
    </trigger>
    <contextcondition>
      $beliefbase.reservation_confirmed || $beliefbase.reservation_rejected
    </contextcondition>
  </plan>
</plans>

```

The plans are not applicable only for the event or belief change. The preconditions and context conditions can be used. The precondition is evaluated before a plan is instantiated and when it is not fulfilled, this plan is excluded from the list of applicable plans. In contrast, the context condition is evaluated during the execution of a plan and whenever it is violated, the plan execution is aborted. Both conditions can be specified in the corresponding tags supplying some boolean Jadex expression. The previous XML code shows how to execute a "reserve_car_by_phone_call" plan. The customer agent is continue executing the plan as long as the agent believes that the reservation is still not confirmed nor rejected. The context conditions in Jadex represent the postconditions of the plan.

6.6.1.2 Agent Capabilities

Different agents often need to use the same or similar functionalities that incorporate more than just a plan behavior. Often a set beliefs and goals are part of a common functionality of one agent. The capability allows for packaging a subset of

beliefs, plans and goals into an agent module and reuses this module wherever needed. The capability structure of an agent forms a tree. A superordinated (parent) capability may contain an arbitrary number of sub-capabilities.

6.6.1.2.1 Creating a Capability

Create a new file **manage_reservation.capability.xml** with the following XML code. The definitions of imports, plans, beliefs, events related with this capability are all placed into this file.

```
<!-- Manage reservation capability -->
<capability xmlns="http://jadex.sourceforge.net/jadex"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://jadex.sourceforge.net/jadex
http://jadex.sourceforge.net/jadex-0.96.xsd"
name="Manage_reservation_capability"
package=" carrental.app ">

    <plans> ... </plans>
    <Beliefs> ... </Beliefs>
    <events> ... </events>
</capability>
```

Modify the agent ADF (*Car_rental_clerk.agent.xml*) by removing all plan and belief definitions. Instead insert a new section for using the new capability.

```
<capabilities>
    <capability name="manage_reservation"> file=" Manage_reservation" </capability>
</capabilities>
```

Note that here the type name is employed, but absolute and relative paths to (the model name of) the XML file can also be used.

6.6.1.3 Events

Jadex is an event-based system, that is, nothing happens inside a Jadex agent unless it is triggered by some event. For example, the *request_information* plan of the car rental clerk agent is triggered when a reservation request message arrives. However, Jadex agents are not purely reactive, because Jadex not only supports external events (e.g., messages), but also different types of internal events. For example, the adoption of a new goal will generate a goal event, leading to plans being executed to achieve the goal. Three types of events are supported in Jadex: Goal events, internal events and message events. More details are present in the Jadex web page. We will present an example how event messages are handled with in the ADF. The following XML code illustrates some event messages related to the car rental clerk agent.

```
// the following events for car rental clerk agent
<events>
    <messageevent name="request_reservation" direction="receive" type="fipa">
        <parameter name="performative" class="String" direction="fixed">
            <value>SFipa.REQUEST</value>
        </parameter>
        <parameter name="content" class="String" direction="fixed">
            <value>" Reserve group B car for rent "</value>
        </parameter>
    </messageevent>
</events>

// the following events for customer agent
<events>
    <messageevent name="agree" direction="receive" type="fipa">
        <parameter name="performative" class="String" direction="fixed">
```

```

        <value>SFipa.AGREE</value>
    </parameter>
    <parameter name="content" class="String" direction="fixed">
        <value>"More information are needed"</value>
    </parameter>
</messageevent>
</events>

```

6.6.1.4 Agent Services Publication

We make the services of our car rental clerk agent publicly available by registering its service description at the Directory Facilitator (DF). Include the DF capability in the ADF to be used:

```

<capabilities>
  <capability name="dfcap" file="jadex.planlib.DF"/>
  <capability name="manage_reservation" file="Manage_reservation"/>
</capabilities>

```

Create a reference for the *df_keep_registered* goal to make it locally available:

```

<goals>
  <maintaingoalref name="df_keep_registered">
    <concrete ref="dfcap.df_keep_registered"/>
  </maintaingoalref>
</goals>

```

Create a configurations section with one configuration. In this configuration, an initial goal for the DF registration should be provided. The agent description that is used for the registration is provided as initial value of the “description” parameter of the *df_keep_registered* goal:

```

<configurations>
  <configuration name="default">
    <goals>
      <initialgoal ref="df_keep_registered">
        <parameter ref="description">
          <value>
            SFipa.createAgentDescription(null,
            SFipa.createServiceDescription("service_reservation",
            " Reply to a customer inquiries ", "EU-Rent"))
          </value>
        </parameter>
        <parameter ref="leasetime">
          <value>non-stop</value>
        </parameter>
      </initialgoal>
    </goals>
  </configuration>
</configurations>

```

Note that when you want to register the agent at a remote DF you only need to slightly modify your initial goal description by adding parameter values for the DF agent identifier and address. Using DF allows our scenario to become a real multi-agent system. The real customer wishes to reserve a car for rent and sends its requests via the message center to the customer agent. The customer agent searches for the agent who offers a reservation service at the DF and subsequently sends a reservation request to the car rental clerk agent.

6.7 Chapter Summary

In this chapter, the entire detailed process of developing multi-agent systems using a case study of the car rental system was discussed. Detailed descriptions of how the car rental system works, which represent the case study to test and evaluate the new methodology, was developed.

PART THREE

EVALUATION AND CONCLUSION

The third part of this thesis provides an evaluation of some agent-oriented methodologies with the new developed methodology and then summarizes the conclusions.

Firstly, the MASD methodology and a few existing agent methodologies are evaluated using the MASADF evaluation framework. This is followed by a comparison of the results. This step is described in chapter 7. Secondly, chapter 8 presents the thesis contribution and deficiencies. The conclusion of the thesis and future work is then presented.

CHAPTER SEVEN

A FRAMEWORK FOR THE EVALUATION OF AGENT ORIENTED METHODOLOGIES

7.1 Introduction

This chapter presents a framework for evaluating agent-oriented methodologies. Four methodologies are evaluated and compared by performing a feature analysis. This is carried out by evaluating the strengths and weaknesses of each participating methodology using an evaluation framework called the Multi-agent System Analysis and Design Framework (**MASADF**). This evaluation framework addresses several major aspects of an agent-oriented methodology, such as: concepts, Models and process. This chapter is organized as follows: section 7.2 describes the evaluation framework MASADF; section 7.3 describes the evaluation results according to the MASADF evaluation framework; and finally, section 7.4 concludes this chapter. The framework should help developers pursuing system development and it also helps developers of agent oriented methodologies to evaluate and improve their methodologies.

7.2 The Evaluation Framework

As was mentioned in chapter 3, many agent-oriented methodologies have been proposed. Even though existing methodologies are based on a strong agent-oriented basis, they need support for essential software engineering issues such as accessibility and expressiveness. This has an adverse effect on industry acceptability and adoption of the agent technology. Therefore, the consequences expected by the agent paradigm cannot yet be fully achieved. Moreover, comparing and selecting agent-oriented methodologies is difficult as they usually address different properties of software agents and methodological aspects. As a result, a comparison framework for the evaluation of those methodologies is needed in order to show their advantages and disadvantages. This framework is an important factor in their improvement and development. This framework sets up on existing work that compares object-oriented (OO) methodologies [Berard 1995; Bobkowska 2005; Prasse 1998; Rumbaugh 1996; Hong 1993] and AOSE methodologies [Henderson-Sellers and Giorgini 2005; Sabas, Delisle and Badri 2002; Cernuzzi and Rossi 2002; O'Malley 2001; Sturm and Shehory 2003]. By including a range of issues that have been identified as important by a range of authors, we avoid a biased comparison. The assessment of each methodology was done by postgraduate students who developed their own designs for the same application using different methodologies and collected comments from each other while they developed their application designs. The aim was to avoid any particular bias by having a range of viewpoints. However, we have customized these criteria to the domain of MAS development. One element of originality in our framework is the use and adaptation of concepts from object-oriented software engineering to the development of MAS methodologies.

The motivation behind this framework is that existing methodologies fail to represent industrial development of MASs. Evaluating methodologies' strengths and weaknesses plays an important role in improving them and in developing a new generation of methodologies.

Our study of the different agent methodologies [Cernuzzi and Rossi 2002; Dam and Winikoff 2002-2003; Juan, Pierce and Sterling 2002; Sabas, Delisle and Badri 2002; Shehory and Sturm, Dori and Shehory 2003; Silva 2004; Sudeikat et al. 2004] indicated that the process of evaluating the methodologies and of making comparisons between them is not an easy task. It requires a large amount of relevant detail in order for the evaluation and comparison to be meaningful, correct, and accurate. An evaluation framework was created and named MASADF. The framework consists of several criteria, which were deduced after detailed studies were carried out. The study looked at common features among the different methodologies in building an agent and how the agent behavior is captured. The MASADF criteria consist of a number of important factors upon which the analysis and design of agents systems depend. These criteria are stated as follows: Models related criteria, process related criteria and supportive related criteria. Each criterion consists of several factors.

7.2.1 Models Related Criteria

Models are a representation that enables capturing the essence of a problem or design in such a way that the translation or mapping from that model to another form can be done without loss of detail. One of the most basic factors that affect the evaluation of a methodology is the group of models of which it consists of, and depends upon. It also includes the level of understanding of those models and the ease with which they can be described, imagined, and visualized for a proposed system. Moreover, the level of complexity will have a big effect on its acceptance and success in research and application. Is it simple or complex to understand? Is the number of models, which make up the methodology so large that the interrelationships between them become too complex?

Whenever the number of models is large, the methodology is more complex in its applicability, learning, and proficiency. The large number of models is also time consuming. In addition, many of the models may lead to some repetition. On the other hand, the lack of models may cause weaknesses in the description of the system, and may lead to the absence of some important essentials to describe the system.

As for the factor “relationships between the models”, this has a direct impact on the ability how these models can be derived. If there is a relationship between the concepts that comprise agents, then it is necessary that there is a relationship between the models that represent these concepts. The absence of such a relationship between models designates a lack of proper distribution of those concepts on the models. This may cause other concepts to emerge that may not be related to the original concepts. This factor “relationships between the models” is also concerned with the level of derivation of models from other models. In other words, is there a clear relationship, which connects or relates between the different models? Is it possible to extract some of the models from others with relative ease? Alternatively, do gaps and splits exist between those models to the extent that it becomes difficult to connect or follow up? This factor is also concerned with how models meet the requirements of the methodology phases. Factors that effect models can be summarized as follows:

1. **Agent Concepts:** this factor is concerned with determining whether the evaluated methodology adheres to agent and multi-agent system concepts. The agent and multi-agent concepts cover the operational and structural aspects. Operational aspects are defined as those that affect the behaviour of an agent and the multi-agent

system. Structural aspects are defined as those that describe the building blocks of an agent and the multi-agent system. Agent-oriented concepts are very important for agent-oriented methodologies in general and for agent-oriented modelling languages in particular. The operational concepts are stated as follows:

Autonomy: The agent's ability to make its own decisions on which action it should perform with minimum intervention of humans or other agents. Does the methodology support describing an agent's self-control features? Does the methodology support modelling a decision-making mechanism of agents regardless of the environment?

Reactivity: The agent's ability to respond in a timely manner to changes in the environment in which it resides. Does the methodology provide mechanisms to represent changes in the environment, e.g. events, incidents, etc.? Does the methodology provide mechanisms to specify and represent agents' responses to changes in the environment?

Pro-activeness: The agent's ability to pursue its own goals over time. Does the methodology provide a goal modelling technique, capturing the system's goals and the agents' goals? Does the methodology provide plans and/or a tasks model, which describe how goals are achieved by an agent?

Sociality: The agent's ability to cooperate with others (humans and agents). How is cooperation supported by the methodology? What cooperation modes are supported by the methodology?

Adaptability: The agent's ability to be situated in an environment. Is an agent able to perceive the environment via its sensors and to initiate actions to affect it?

Mental notions (such as beliefs, goals, and plans): Does the methodology support modelling mental notions of agents? For agents modelled in the methodology; are they able to store information about their environment and their internal states as well as the actions they may carry out?

Relationship: This represents the dependency relationships between the agents. Does the methodology support modelling dependencies between agents? For agents modelled in the methodology do they depend on each other to achieve goals?

Communication: Interactions between agents are mainly achieved via communication. What communication modes are supported by the methodology? Is it synchronous or asynchronous?

2. **Agent Attributes:** This factor is concerned with the description of the internal structure and the parts that make up the agent. The important attributes are defined and must be approved as a base to enable an agent to be independent and yet capable to adapt to and interact with the surrounding environment. An agent must also be capable to satisfy its need, make use of its interests and control its beliefs.
3. **Ease of Use and Ease to Learn:** It is important for models of the methodology to be easy to use and easy to learn. Does the model contain notations that are well known and easy to learn by both experts and novice developers? Are they easy to

apply? Is it easy to represent the models provided by the methodology? This factor measures the ease and the simplicity of understanding the system to be created in its entirety. Simplicity refers to the ease of use. It is important for a methodology not only to be understandable to the users but also to be easy to use. The first step towards using a methodology is to learn the notation. Hence, it is desirable that the notation be easy to learn by both experts and novice users (Rumbaugh 1996). In addition, the easier the users can remember the notation, the quicker they are able to learn to use it. Therefore, the notation should be as simple as possible. Is it easy to draw and write by hand the notation provided by the methodology? Are the diagrams produced using the methodology clear? Are the techniques used to understand and describe the system behaviour clear?

4. **Visualization Ability:** Since developers usually draw models by hand during the process of reviewing designs, it is essential for the notation to be easy to be written by hand. It is also important that the diagrams produced by the methodology should be easy to read. Does the notation contain symbols that are familiar to users and easy for them to remember?
5. **Expressiveness:** This factor measures whether the models of the methodology represent all the necessary aspects of a system in a clear and natural way. How well can each model express the concepts (e.g. is each model capable of capturing the concept at a great level of detail, or from different angles)? Is the notation capable of expressing models of both static aspects of the system and dynamic aspects? Static aspects are those that represent relationships such as aggregation, specialization, structure of the system, and the knowledge encapsulated within the system. Dynamic aspects describe the processing, agent interaction, stage changes, timing, and data control flow within the system.
6. **Consistency:** Models should not contradict each other. Does the methodology provide guidelines and techniques for consistency checking both within and between models?
7. **Traceability and Model Derivation:** This represents the relationships between models and between models and the requirements of the system. Traceability requires that it has to be easy for the designers to understand and trace through the models. This may increase the users' understanding of the system. Is there a clear and easily recognizable path from early analysis to implementation via different modelling activities?
8. **Refinement:** This is a way of developing a design as it allows the developers to improve design artifacts at different points in the development process. Therefore, it is worthwhile that a methodology should provide mechanisms to support refinement. Is the modelling language integrated into the development process? Can a model be built incrementally? For instance, the designers can start from the most abstract level to subsequent levels of detail. Is there a seamless transformation from one level of abstraction to another without causing the loss of semantics?
9. **Ability to model Agent Interactions:** Agents in multi-agent systems need to communicate in order to plan and coordinate the work that needs to be carried out. This factor is concerned with the ability of the methodology to model and represent

the communication between agents in the system. It is also concerned with defining the techniques and the protocols used in performing such communication.

7.2.2 Process Related Criteria

In this section, aspects related to the development process of MASs (how they are built) will be evaluated. This section looks at the applicability of the methodology, the stages provided for its development process, and the development approach followed by the methodology. In particular, to achieve this evaluation the following aspects must be considered:

1. **Development lifecycle:** What development lifecycle describes the methodology best? Software engineering approaches propose two types of methodology development: lifecycle models (such as waterfall) and iterative.
2. **Coverage of the lifecycle:** What phases of the lifecycle are covered by the methodology? The ideal methodology must cover six important stages, which are: system requirements, analysis, architectural design, detailed design, implementation and testing.
3. **Development perspective:** What development perspective is supported (i.e. top-down, bottom-up, or hybrid)?
4. **Application domain:** Is the methodology applicable to a specific or multiple application domains?

7.2.3 Supportive Feature Criteria

Supportive feature criteria evaluate a variety of high-level, supplementary features of the methodology. These features include the availability of software and methodological support, dynamic structure support, and open systems support.

1. **Software and methodological support:** This factor measures the maturity of a methodology and can play an important role in determining the quality of it. There are two ways to measure the maturity of a methodology:
 - What are available resources supporting the methodology? For example, conference papers, journal papers, text book, tutorial notes, consulting services, training services, etc.
 - Is the methodology supported by tools? For example, supporting tools can be tools for building models, project management, rapid prototyping, reverse engineering, automatic testing, and diagram editors, code generators and design consistency checkers.
2. **Open systems support:** This factor tries to evaluate if the methodology is intended for open systems or not, (i.e. it allows the dynamic addition or removal of agents, or their characteristics, while the MAS is running).
3. **Robustness support:** Does the methodology provide support for robustness (e.g. the methodology provides techniques to analyze system performance for all configurations, or provides techniques to detect/recover from failures)?

4. **Mobility support:** Does the methodology cater for the use/integration of mobile agents in MASs (e.g. Does the methodology allow the developer to model which/when/how agents should be mobile)?

7.3 Evaluation Results

This section presents results of a detailed evaluation of a number of methodologies and makes a comparison thereafter. The evaluation is performed using the proposed criteria mentioned above. A comparison is presented of the following four methodologies: Gaia, MaSE, HLIM and our approach MASD. The results are discussed below. The assessments for criteria are summarized in tables 7.1, 7.2 and 7.3 whereas those for “narrative” criteria are discussed in detail in text. It is also noted that the discussion of the results is structured in terms of criteria presented in section 7.2.

7.3.1 Models Related Criteria

7.3.1.1 Agent concepts

Autonomy: according to our evaluation, the four methodologies recognize its importance. The level of support for autonomy in all of them overall is “good” (ranging from medium to high). This is reflected by the fact that all the four methodologies provide various supports for describing agents' self-control features. For instance, functionalities and tasks are encapsulated within an agent. In addition, roles model in Gaia, concurrent task diagrams in MaSE, goals model in HLIM or goals model and roles model in MASD allow the decision-making mechanism of agents to be modelled regardless of the environment and other entities. That mechanism is based upon the agents' goals and their roles within the system. Each agent playing a role has its own autonomy of decision and its behaviours are not fully controllable or predictable, but are regulated by its responsibilities into the organization [Yu 2002]. Since Gaia, MaSE and MASD strongly support role definition, they also support autonomy. Despite the fact that HLIM does not support any type of roles, it nevertheless provides some degree of autonomy by supporting goals and plans definitions.

Reactiveness: in terms of support for reactivity, the evaluation is slightly different. Gaia and MASD support reactivity concept. Gaia supports some degree of reactivity through the liveness properties within the role's responsibilities. However, this does not specify the occurrence of events and the role's reaction to these events. MASD strongly supports reactivity through the triggers concept or model, which motivates the agent to reacting to events that occur during runtime according to these triggers. In MaSE and HLIM the reactivity concept is not expressed explicitly. That is, there is no explicit connection between the event and the action taken.

Pro-activeness: in terms of support for proactiveness, the assessment is varying. In Gaia, the proactiveness is expressed by the liveness properties within the role's responsibilities. In MaSE, the proactiveness is expressed by the role's tasks. These tasks are modelled using finite state automation. MASD and HLIM support proactivity through pursuing goals. Goals are important to agent-based systems because agents are autonomous and proactive. Agents achieve goals on behalf of users through their autonomous and proactive behaviour.

Sociality: in GAIA, sociability is expressed through the acquaintance model in which the agent types' interactions are depicted. Further, its sociality is expressed using the organizational structure and roles. In MaSE and HLIM, social aspects (except for communication and conversations) are not mentioned. MASD supports some degree of sociality by providing the interactions between these agents as well as the roles that are to be performed by the agents. This provides agents with the ability to interact with each others and humans through the interaction model, which describes the interaction mechanisms.

Adaptability: in terms of support for adaptability, the assessment is also varying. Gaia, MaSE and HLIM do not support adaptability. MASD supports some degree of adaptability through the triggers model, which gives the agent the ability to perceive the environment and to initiate actions to affect it.

Mental attitudes: Gaia does not support the use of mental attitudes (such as beliefs, desires, and intentions). MaSE provides weaker support for capturing an agent's mental attitudes. MaSE provides goal diagrams but it does not have a representation of the agent's beliefs. HLIM and MASD certainly provide better support than MaSE for capturing an agent's mental attitude. They also have goal diagrams and they have a representation of the agent's beliefs. MASD represents the agent knowledge of the world as beliefs. In MASD, desires are represented as goals without plans. Intentions are represented as goals with predefined plans.

Relationship: Gaia and MaSE do not support any type of relationships between the agents. HLIM and MASD support the relationships by providing the dependency relationship diagram, which helps the agents to make the necessary decisions when cooperation between these agents takes place.

Communication: Gaia supports communication by its own interaction protocols. In terms of support for communication, MaSE is probably best with its protocol analyzer. HLIM supports communication between agents through its own protocols. This is despite the fact that protocols are not clearly supported by HLIM. MASD supports communication by the inter-agent communication model. MASD supports FIPA-ACL protocols.

7.3.1.2 Agent Attributes

The Gaia methodology generally differs from all other methodologies. More specifically, it differs from the above-mentioned methodologies in the internal structure of the agent. The Gaia methodology depends on the "Role" during the analysis phase as a base for the agent. During the design phase, roles are replaced with agents. It can be stated that the internal structure of the agent in the Gaia methodology is somehow strong. The strength is due to the fact that every agent plays a specific role and is independent in making decisions. On the other hand, the agent behavior is not fully controlled but it can be controlled through responsibilities inside the organization.

In MaSE, the agent's internal structure is formed through the step of assembling agent classes during the design phase. Designers have the freedom of choice in whether they wish to use an architecture that they developed, or to use a predefined and available architecture such as Belief-Desire-Intention (BDI), reactive, planning, knowledge-base,

and user-defined agent architecture. This makes the methodology flexible and able to be utilized with different architectures.

In the HLIM methodology, the agent's internal structure is represented and described by a number of properties like goals, plans, and beliefs. It does not support the BDI architecture.

The internal structure of the agent in MASD is described very comprehensively. MASD methodology strongly supports the BDI architecture. Where the attributes that enable the agent to be able to adapt and interact with its environment are supported such as agent roles, goals, plans, beliefs, triggers and dependencies, etc.

7.3.1.3 Ease of Use and Ease to Learn

With respect to the ease of use, the Gaia methodology is found to be good in this respect in some models only. For example, clear one-to-one mappings and direct relationships were found between some models. The transfer method between the roles diagram and the agents diagram is also clear and straightforward. However, some models had no relationships between them. Some models and charts that Gaia uses are relatively good and some are insufficient. For example, the role model is clear and can easily be understood. While the interactions model is weak and does not carry out the task it is supposed to do well when compared with other methodologies.

The MaSE methodology is largely complex in organizing and ordering the diagrams and charts based on the methodology. The concurrent task diagram is derived from a number of charts: pyramid goal chart, sequential chart and role chart. This method may cause some confusion and ambiguity to the analyst. In comparison with the other methodologies, Gaia, HLIM and MASD have easier, more flexible and more direct ways in deriving models. MaSE also supports one-to-one mappings between some models such as the goals model to the roles model and from the roles model to the agent class model.

The HLIM methodology is reasonably good with respect to the ease of understanding and visualizing the system's models. It uses easy to understand technologies like the use case maps to describe an agent's behavior in a system. There is also no complexity in the models. They can be easily and flexibly extracted from one another. Relations can also be directly recognized. On the other hand, with respect to the coverage of all relevant phases, it lacks the detailed design phase. Thus, there exists a gap between the design and implementation phase.

The MASD models and charts are relatively good and sufficient. For example, the system scenario model is clear and easy to understand. Concerning the ease of use, MASD models are easy to use as well as easy to learn. They contain notations and techniques (such as UCMS, UCDs, UML Activity diagrams, FIPA-ACL) that are well known to developers and easy to learn to construct and apply them and it is easy to represent them.

7.3.1.4 Visualization Ability

In Gaia understanding and visualizing the system requirements is not included explicitly. Gaia lacks the existence of an initial, and a primary phase, which is the initial requirements phase. Description techniques for describing the whole system behavior

are not supported. It lacks techniques and tools like the High Level Message Sequence Charts or the Use-Case Maps, which are known for high-level visual display of system descriptions. The system description comes in the form of diagrams or notations that help to understand the systems.

In the MaSE methodology, understanding and visualizing the system requirements is included in the second stage of the analysis phase. The Use-Case is used which contains summaries of the main message exchanges in the initial flow of the system. To a certain extent, this method is not sufficient in describing the system behavior. The reason is that Use Cases are only used in the requirements analysis phase in order to help the customer to understand the system structures from his point of view alone.

In the HLIM methodology, understanding and visualizing the system requirements is actually included in its discovery phase. It is represented in the high-level model, which uses the Use-case maps notation. It recognizes the agents from other components in the system and their high-level behavior. HLIM provides a high view of the system and provides a starting point to develop other model details in the system.

In the MASD methodology, understanding and visualizing the system requirements is represented by using some well-known techniques such UCMs and UCDs. These techniques are very simple and intuitive. They have the ability to visualize the complex system requirements as scenarios in one model. With regard to the visualization ability, the existence of an initial and a primary phase, which is the initial system requirements phase, helps the developers capturing a high level view of the whole system. Understanding and visualizing the system is contained in the system requirement phase where UCMs and UCDs are used. UCMs recognize the agents from other components in the system and their high level behavior. It provides a high-level view of the system and provides a starting point to develop other model details in the system. UCDs help the customer to understand the system structures from the user's point of view.

7.3.1.5 Expressiveness

In terms of support of expressiveness, the number of static and dynamic aspects is a good indicator of this criterion. Gaia, MaSE and MASD are capable of expressing models of both static aspects and dynamic aspects of the system. Gaia supports the dynamic aspects of the system and handles protocols well by providing an interaction model with its own interaction conversation protocols. MaSE supports dynamic aspects by providing a communication class diagram through a finite state machine. MASD supports dynamic aspects by providing Interaction diagrams and FIPA-ACL protocols. HLIM clearly does not support dynamic aspects and protocols.

7.3.1.6 Consistency

In terms of consistency checking, the level of support differs from one methodology to another. It is well supported in MaSE whereas is not supported in Gaia, HLIM and MASD. This result seems to be related to the tool support integrated with the methodology. AgentTool (developed by MaSE) provides a strong support for model and design consistency checking. The remaining three methodologies do not have any tool at all (Gaia, HLIM and MASD).

7.3.1.7 Traceability and Model Derivation

Gaia does not explicitly support traceability and model derivation, however MaSE, HLIM and MASD support this feature. There are clear links between models provided by them. For instance, goals, roles, agents, and tasks are all linked together. This strong connection improves the ability to track dependencies between different models. Such connections allow developers to (automatically or manually) derive design models (e.g. an agent's internal architecture) from analysis constructs. MASD is good in this respect. The models are traceable and can be derived easily from each others, and they have the ability of mapping or transferring from one model or diagram to another. Most of the models in MASD are derived from the system scenario model.

7.3.1.8 Refinement

Refinement is generally well supported by all four methodologies. This reflects the fact that the modeling language of all four methodologies is integrated into their development process. The process in fact consists of iterative activities. Developers are free to move between phases to add more detail in a constructed model. Another indication of refinement supported by the four methodologies is the seamless transformation from one level of abstractions (e.g. goals, roles) to another (e.g. agents, tasks) without causing loss of semantics.

7.3.1.9 Ability to Model Agent Interactions

Collaboration protocols between agents in the Gaia methodology are insufficient and require more development and improvement. Gaia only supports one-to-one interactions between agents. It does not support simultaneous interactions between multiple agents.

The MaSE methodology describes conversations between agents through finite state machines. It is possible through this to achieve a dynamic level of message exchange between agents in a system. These finite state machines lead to an algebraic description of conversations. Official mathematical proofs can be formulated from this algebraic description to describe and proof the interaction between agents in a system. For this reason, this method is considered successful and acceptable in describing the interaction between agents.

HLIM does not support a detailed design phase. It describes conversations between agents through a conversation model, which is extracted from the agent relationship model and the internal agent model. This means that this model defines all messages between agents in the system. The messages implement dependency relationships as recognized in the agent relationship model. However, the messages are defined on a high general level, not in detail. The detailed interactions are an important outcome of the detailed design phase. Hence, this methodology lacks a detailed design phase in which the detailed interaction protocols can be specified.

MASD describes the interaction between agents through the interaction model. These interactions are sufficient because they are based on FIPA-ACL protocols, which are considered as standard.

The results of the evaluation of the four methodologies with respect to model related criteria are shown in table 7.1. Each column in the table represents a particular methodology. The first column lists the factors. Each methodology column represents the supporting state of that factor. The assessment scale has four possible answers as follows: “high”, “medium”, “low”, and “none”. These answers indicate the level of support of the methodology for a particular factor.

Criterion	Gaia Methodology	MaSE Methodology	HLIM Methodology	MASD Methodology
Concept				
<i>Autonomy</i>	High	High	Medium	High
<i>Reactivity</i>	Medium	Low	Low	High
<i>Proactivity</i>	Low	High	Low	Medium
<i>Sociality</i>	High	Low	Low	Medium
<i>Adaptability</i>	None	None	None	Low
<i>Mental notions</i>	None	Low	High	High
<i>Relationships</i>	None	None	High	High
<i>Communication</i>	Medium	High	Medium	High
Agent attributes	Medium	High	Medium	High
Easy to use and Easy to learn	Medium	High	High	High
Visualization ability	Medium	High	High	High
Expressiveness	Medium	High	Low	High
Consistency	None	Medium	None	None
Traceability and model derivation	Low	High	High	High
Refinement	High	High	High	High
Ability to model agent interactions	Medium	High	Medium	High

Table 7.1 Evaluation by Models Related Criteria.

7.3.2 Process Related Criteria

7.3.2.1 Development lifecycle

With respect to the development lifecycle factor, all four methodologies are considered as iterative across every phase. All of these methodologies support the model refinement process, which in fact consists of iterative activities. These iterative activities allow developers to move between phases to add more details in a constructed model.

7.3.2.2 Coverage of the lifecycle

To a certain extent, all methodologies are acceptable at varying levels when it comes to the important factor of the life cycle. For example, the methodologies Gaia, MaSE and HLIM cover the fundamental phases like the analysis and design phases, but Gaia and HLIM lack a detailed description of the implementation. The HLIM methodology needs improvement and development in the detailed design model. None of the methodologies supports the testing phases. The MASD methodology covers all phases in the software development lifecycle (except testing phase). The testing phase is not supported by MASD.

7.3.2.3 Development Prospective

With respect to the development prospective factor, all four methodologies are a top-down approach.

7.3.2.4 Domain Applicability

With regard to the development domain applicability, all four methodologies are considered independent. They can be applied in any application domain.

The results of the evaluation of the methodologies with respect to process related criteria are shown in table 7.2. Each methodology column represents the supporting state of that factor. The assessment scale has different possible answers as follows: “iterative” or “waterfall” for the development lifecycle criterion, “yes” and “no” for the coverage of the lifecycle criterion, “top-down” or “bottom-up” for the development prospective criterion and “independent” or another specific application name for the domain applicability. These answers indicate the support that the methodology provides for a particular factor.

Criterion	Gaia Methodology	MaSE Methodology	HLIM Methodology	MASD Methodology
Development Life cycle	Iterative	Iterative	Iterative	Iterative
Coverage of Life cycle				
System Requirement	No	Yes	Yes	Yes
Analysis	Yes	Yes	Yes	Yes
Architectural design	Yes	Yes	Yes	Yes
Detailed design	No	Yes	No	Yes
Implementation	No	No	No	Yes
Testability	No	No	No	No
Development prospective	Top-down	Top-down	Top-down	Top-down
Domain applicability	Independent	Independent	Independent	Independent

Table 7.2 Evaluation by Process Related Criteria

7.3.3 Supportive Feature Criteria

The supportive related criteria consist of a number of important factors as follows:

7.3.3.1 Software and Methodological Support

Regarding the availability of resources supporting the methodologies, Gaia, MaSE and HLIM are covered by conference papers, journal papers or technical reports. MASD is the newest methodology and therefore is not yet available. None of the four methodologies are published as textbooks. The availability of tool support also varies. MaSE is well supported with AgentTool (MaSE). According to the authors of MaSE, AgentTool can be used as a diagram editor, a design consistency checker, code generator and automatic tester. They also revealed that AgentTool has been downloaded and used by many people in academia as well as industry and government. Gaia, HLIM and MASD do not provide any support tools.

7.3.3.2 Open Systems Support

Regarding the open systems support criterion this issue is, in our point of view, not explicitly addressed in the MaSE, HLIM and MASD methodologies. More specifically, they do not deal with the introduction of new components or modules in an existing system. Gaia supports open systems development by providing the appropriate organizational abstractions that are central to the analysis and design of open multi-agent systems. MASD is intended to work for cross-boundary systems (semi-open systems) where the agent system itself is closed (i.e. the types and behaviours of agents defined in the system are determined in advance). The external agents may interact with the system agents through predefined established protocols (e.g., FIPA).

7.3.3.3 Robustness Support

Concerning the robustness support, none of the four methodologies address this factor. More specifically, they do not provide any techniques to deal with analyzing the system performance. Furthermore, none of the methodologies supports techniques to recover the agent system from failures.

7.3.3.4 Mobility Support

Regarding the mobility support criterion, in our opinion, this issue is not explicitly addressed in any of the methodologies. None of the four methodologies mention mobility issues.

The results of the evaluation of the four methodologies with respect to support related criteria are shown in table 7.3. Each methodology column represents the supporting state of that factor. The assessment scale has two possible answers “yes” and “no”. These answers indicate the support of the methodology for a particular factor.

Criterion	Gaia Methodology	MaSE Methodology	HLIM Methodology	MASD Methodology
Software and methodological support	No	Yes (AgentTool)	No	No
Open systems support	Yes	No	No	semi-open
Robustness support	No	No	No	No
Mobility support	No	No	No	No

Table 7.3 Evaluation by Supportive Related Criteria

7.4 Discussion

In the previous sections, our evaluation analysis of the four selected agent-oriented methodologies has been looked at. The methodologies are currently, in our view, among the most commonly known agent-oriented methodologies. Their strengths and weaknesses have been assessed based on a feature-based evaluation. In addition, similarities and distinguishing differences with respect to their techniques and models were also examined.

Overall, each of the four methodologies provides reasonable support for the models related criteria and process criteria. Most of the methodologies partially address the supportive related criteria. MaSE supports the software and methodological support factor. Gaia supports the open systems factor. MASD supports only semi-open systems. These methodologies are also regarded by their developers and the students as clearly agent-oriented. However, there are several characteristics of agent-based systems that are not addressed or sufficiently addressed in most of the methodologies. For instance, none of the four methodologies provide explicit support for agent adaptability, testability, open systems, robustness and agent mobility. Methodologies that are most complete in terms of their support for models related criteria are Gaia, MaSE and MASD. The methodologies that support some factors for process related criteria are Gaia, MaSE and MASD.

As can be seen from the previous evaluation, a relatively complete and comprehensive agent-oriented software engineering methodology should include the following aspects: concepts, models, and processes. In addition, it should at least cover the most important development phases such as: the requirements, analysis, design, and implementation phases. The models and techniques used in these phases can be formed to unify those used in existing agent-oriented methodologies that were examined in this research.

7.5 Chapter Summary

This chapter presented a comparison between three well-known methodologies and the MASD methodology with respect to the construction and development of agent systems. The chapter also discussed the foundational concepts for these methodologies. The MASADF criteria were used to evaluate the methodologies concerned with the building and development of agents systems.

CHAPTER EIGHT

CONCLUSION

This chapter provides the conclusions of this thesis. Some things worked as planned and some did not. In general, the MASD methodology has a larger scope and correspondingly less depth than anticipated. This chapter is composed of four sections as follows: Contributions, deficiencies, conclusion and future work.

8.1 MASD Advantages

Developing and constructing a complete methodology is not an easy task. The MASD methodology is developed as a step towards a comprehensive version. MASD is constructed based on a well-defined, structured set of aspects that an agent-oriented methodology should include. These aspects are: the entire set of guidelines and activities, a full lifecycle process, a comprehensive set of concepts, modeling techniques, and process.

The new proposed MASD methodology should help to improve the MAS development process. The proposed methodology is to be distinguished from existing methodologies in several aspects:

- 1) The MASD methodology is based on correct concepts where the concepts were selected and chosen to be a solid foundation for the building of the new methodology.
- 2) The MASD methodology supports several important features such as: flexibility, consistency, simplicity, and ease of use as well as traceability. This is in contrast to the difficulty of understanding and implementing existing methodologies, resulting in a lack of success.
- 3) The proposed methodology covers the fundamental phases as a full software development lifecycle for building systems. The operations starts at the system requirements phase and extends to the implementation phase. Most of the existing methodologies suffer from the problem of incompleteness.
- 4) The proposed methodology covers the most important characteristics of multi-agent systems. It deals with the agent concept as a high-level abstraction capable of modeling the complex system.
- 5) The new methodology incorporates well-known techniques for requirement gathering and customer communication. It goes further by linking them to the domain analysis and design models. It also supports high-level designs and describes the functional requirement of the system from an external perspective.
- 6) The new methodology supports agent organizational aspects. An agent organization is a group of agents working together to achieve a common purpose. It consists of roles that characterize agents. By utilizing these roles, the methodology allows developers to work at different levels of abstraction. Agent behaviour can be specified at both the level of roles and at the level of role characteristics.
- 7) The methodology proposes a new concept called the “trigger concept” which is considered as one of the most important characteristics that represent the agent

autonomy and reactivity. The existence of the trigger concept plays an important role in determining a larger part of the behaviour of the agent.

- 8) The new methodology presents a clear understanding of MASs and how to build them without referring to implementation detail. It sets the distinctive characteristics of a MAS as a system that has its own structure and composition.

8.2 MASD Deficiencies

Several phases of MASD describe mapping from one model to another. In each case, the transformation is accompanied by guidelines for the designer to consider when performing the transformation. Nearly every transformation can be enhanced by the addition of a systematic procedure that details how the transformation must take place. Additionally, the transformations of roles to goals and plans and roles to agent would benefit from a similar process that was focused more on a series of rules than on guidelines. Furthermore, MASD does not provide verification and validation techniques in an application design and, especially, in its implementation phase. Also, MASD does not provide testing phase and it does not provide any tools to support the methodology process.

8.3 Discussion

In this section, we discuss the deficiencies and limitations of existing agent oriented methodologies which we have addressed and solved in this dissertation and those that are not solved as well. The addressed problems are stated first followed by a discussion.

No standard has evolved: Developing and establishing agent standards is not an easy task, because the standardisation process shifts the debate from longer term research issues to the ability to practice commercial agent systems. Therefore, MASD does not propose any standard definitions, nor agent architecture or any aspects related to agent languages which can be considered as a standard.

Industrial development suitability: Although, MASD is still new, it is difficult to assess whether it is suitable for industry or not. However, we assure it will be suitable for industrial development of multi-agent systems and it will be accepted in the industrial domain. That is because of its simplicity, easy to learn and its completion of stages of systems development.

Neediness for formal semantics: Despite the MASD methodology is developed based on concepts chosen in precise manner to cover most of the existing agent definition patterns, MASD does not have any formal semantics.

Existing gap between design and implementation: The MASD methodology bridges the gap between design and implementation. This is achieved by providing refined design models such as an agent container in the design phase. They can be directly transferred into implementation constructs in an available programming language. It provides constructs to directly implement high-level design concepts. Alternatively, a dedicated agent-oriented programming language such as Jadex, JADE, JACK, etc. can be used.

Implementation phase inclusion: The MASD methodology provides an explicit implementation phase and is considered as an essential phase of its process. Most of the existing methodologies do not include an implementation phase. The Jadex framework is used as a programming environment to implement the agent system. Jadex describe in detail how the beliefs, goals, plans, and interactions are implemented, as well as it

explains how to implement reasoning about beliefs, goals, plans and reasoning of communication.

Support of multiple Roles: One of the most important aspects of the MASD methodology is that it considers the role concept as a one of the main concepts that the methodology relies on. MASD assumes agents can play one or more roles at a time. This is achieved by providing the roles model in the analysis phase which describes the roles that the agent will play in the system.

Agent-oriented approaches: According to the agent methodology classification discussed in section 3.4.1 in chapter 3, The MASD methodology is considered to be an agent based methodology. Therefore, the agent (and its internal components) developed by the MASD methodology is built from scratch as an individual entity without dependence on any other traditional methodologies, such as object-oriented methodologies.

Environment features inclusion: The MASD methodology takes into account the environment issues by providing the MAS architecture stage in the analysis phase. This stage describes how to identify relationships and interactions between the entities (agents) that inhabit the environment (MAS), the conversations and exchanged messages and the services that are offered by the agents in the system.

Software engineering issues: the MASD methodology is established based on essential software engineering issues such as preciseness, accessibility, expressiveness, domain applicability, modularity, refinement, model derivation, traceability, and clear definitions. The preciseness issue is represented in MASD by providing well known modelling techniques such as UML UCDs, UCMs, and UML activity diagrams which have clear semantics. The accessibility issue is represented by modelling techniques that are easy to understand and easy to learn such as UCMs and UML UCDs. With respect to the expressiveness issue, the MASD methodology addresses this issue by providing a clear step by step development process. This process describes the whole structure of the system. In addition, MASD supports dynamic aspects by providing interaction diagrams and FIPA-ACL protocols. With regards to domain applicability MASD is independent and it can be applied to any application domain. The MASD methodology addresses the modularity issue by providing organized phases for the MAS development process. Refinement is generally supported by MASD. MASD uses iterative activities which are integrated into its development process. MASD supports both model derivation as well as traceability issues. There are clear links between models provided by MASD. For instance, roles, agents, goals, and plans are all linked together. The MASD models are traceable and can be derived easily from each other, and they provide a mapping from one model or diagram to another. Most of the models in MASD are derived from the system scenario model. MASD is based on clear definitions.

Misconceptions: MASD is established based on precise accepted MAS definitions. These definitions are chosen from the literature to cover most of the existing agent definition patterns.

Incompleteness: MASD is considered as a complete development process. It provides a full lifecycle development process for MAS. This process starts with the initial specification, system requirements, and finally producing implementation code.

Incomplete formality: MASD methodology does not address any formalism of MAS concepts.

Open systems: MASD methodology is not intended to work with open systems. But it is designed to work for cross-boundary systems (semi-open systems) where the agent society itself is closed (i.e. the types and behaviours of agents defined in the system are determined a priori) but external agents may interact with members of the society via defined and common protocols (e.g., FIPA).

8.4 Conclusion

As agent-oriented approaches represent an emerging paradigm in software engineering, there has been a strong demand to apply the agent paradigm in large and complex industrial applications and across different domains. In doing so, the availability of agent-oriented methodologies that support the software engineers in developing agent-based systems is very important. In recent years, there have been an increasing number of methodologies developed for agent-oriented software engineering. However, none of them are mature and complete enough to fully support the industrial needs for agent-based system development.

For all those reasons, it is useful to commence gathering together the work of existing agent-oriented methodologies with the aim of developing a future methodology that is mature and complete. Thus, this research is focused on developing a comprehensive design methodology to assist a multi-agent system designer through the entire software development lifecycle, beginning from the system requirement phase and proceeding in a structured manner towards a working code.

There are few principal strengths of the methodology developed through this research work. First, it is based on three important aspects: concepts, models, and process, and it is focused toward the specific capabilities of multi-agent systems. At the commencement of research, MASD methodology combined several techniques and concepts into a single, simple, traceable, and structured methodology. These concepts and techniques are represented through a set of models. Most of these models used within the methodology have therefore been already justified and validated within the domain of agents and multi-agent systems. MASD provides an extensive guidance for the process of developing the design and for communicating the design within a work group. It was very clear that the existence of this methodology provides a great assistance in thinking about and deciding on design issues, as well as conveying design decisions.

The MASD methodology has captured all the requirements of the system in a proper way by combining well known techniques (UCMs and UCDs) into one extensive model called system scenario model. Moreover, MASD has introduced MAS concepts through conceptual framework where the concepts are determined, and selected. This conceptual framework has been used to introduce the MAS concepts that the new methodology relies on. MASD methodology has proposed the use of the trigger concept which has allowed the representation of agent reactivity. Finally, MASD has proven its ability to support organizational aspects by utilizing the role concept which provides the work at different levels of abstraction.

8.5 Future Work

This section lists several topics that are not addressed in this thesis. Each topic would clearly benefit from further investigation and, hopefully, would make the MASD methodology stronger. Candidate topics for future investigation are:

- How to utilize the methodology with special domains such as web-based application, real time systems, etc? ;
- How to perform testing for the resulting agent system software?, and

- How to deal with issues related to agent project management, such as: metrics, estimation, schedule, risk and quality?
- How to deal with the agent mobility?

APPENDIXES

APPENDIX A: USE CASE MAPS

A.1 Use case Maps (UCMs)

This appendix presents an introduction to use case maps. It concentrates on explaining what UCMs are, and why the technique is needed. A UCM is a high level visual view. It helps individuals to visualize, think about, and explain the overall behavior of a whole system. It also guides the development for the design of high-level architectures and detailed scenarios from requirements [Buhr 1998]. UCMs identify agents and their high-level behaviour.

The UCM technique is growing in popularity. Whether causal scenarios, architectural entities, or behaviour patterns, they help to understand and describe the behaviour of complex and dynamic systems.

The phrase, causal paths cutting across organizational structures, sums up the fundamental meaning of a UCM. This idea produces a lightweight notation that comprises all the foregoing complexity issues in an incorporated and controllable method. The notation allows an easy visualization of the casual paths, which thread through the system, avoiding the intricate details. The casual paths, or behavior structures, symbolize large-scale elements of emergent behavior. Network transactions are an example of the unparalleled architectural entities that cut across these systems and are independent and above the level of details as they can be realized in a different, detailed way.

A.1.1 Where are UCMs Useful?

UCMs are designed to be useful for requirement specification, design, evolution, adaptation, maintenance, and testing. UCMs have been used in the following fields:

- Requirements engineering and design of:
 - Multimedia systems
 - Real-time systems
 - Agent systems
 - Object-oriented systems
 - Distributed systems
 - Telecommunication systems
- Initial phases of development and documentation of standards
- Discovery and evasion of unwanted feature interactions
- Implementation analysis and forecasting
- Assessment of architectural alternatives
- Functional assessments
- Recognition of race conditions
- Production of message sequence charts and official specifications
- Reverse-engineering of diverse systems

Details are not the focal point of UCMs. Scenarios are described in terms of casual relationships between responsibilities. UCMs highlight the most interesting, critical and relevant functions of the system and portray complex systems at a high level of abstraction. The complex system requirements are captured as UCM scenarios integrate into a single model with stubs and plug-ins. Table A.1 shows basic UCMs symbols of the type that are used through out this thesis.”

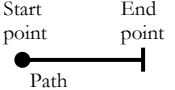







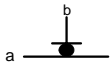

UCM Notation	Notation Explanation
	Path: Represents flow of events in a system; path connects start points, stubs, responsibilities, forks, and end points of UCM. The start-point represents preconditions. The end-point represents post-conditions.
	Responsibility point: Represents the functions to be accomplished by the system at that point of the path.
	Or Fork: An OR fork means the path proceeds in only one out of two or more directions.
	Or Join: it means two or more paths merged it in one single path.
	And Fork: it means that a single path is distributed at the same time into many concurrent paths.
	And Join: it means that several concurrent Paths merged at the same time into a single path.
	Static stub: contains one plug-in (Sub UCM) as task to be achieved by the system, Used as decomposition of complex maps.
	Dynamic stub: May contain several plug-ins, whose selection can be determined at run-time according to selection policy (often described with preconditions). It is also possible to select multiple plug-ins at once (sequentially or parallel).
	Wait point: Path a waits for an event from path b.
	Component representing roles in the system.

Table A.1 Basic Use-Case Maps (UCMs) symbols

Use case maps are used for the following reasons:

- Intuitiveness: can be understood easily by humans.
- Multiple scenarios and the interactions amongst them can be shown in one diagram.
- Ability to map the scenarios into different architectures or (formal) models
- Gives visual representation of scenarios
- Simplicity: They are simple and very easy to draw and do not need tools.

A.2 UCMs by Example

UCMs are considered as precise structural entities. They have within adequate information in a highly condensed form to enable persons to visualize system behavior. It provides a high level view of causal sequences in the system as a whole, in the form of paths. The causal sequences are called scenarios. Generally, UCMs may have many paths [Elammari and Lalonde 1999]. For the purpose of simplicity, figure A.1 UCM scenarios for money withdrawal shows an example of just one such path. This simple example represents “Money withdrawal using an Automatic Teller Machine (ATM)”. The scenario starts with a triggering event or a pre-condition (filled circle labeled Customer wants to withdraw money) and ends with one or more resulting events or post-conditions (bar labeled logon rejected, withdraw rejected, slip printed and money withdrawn).

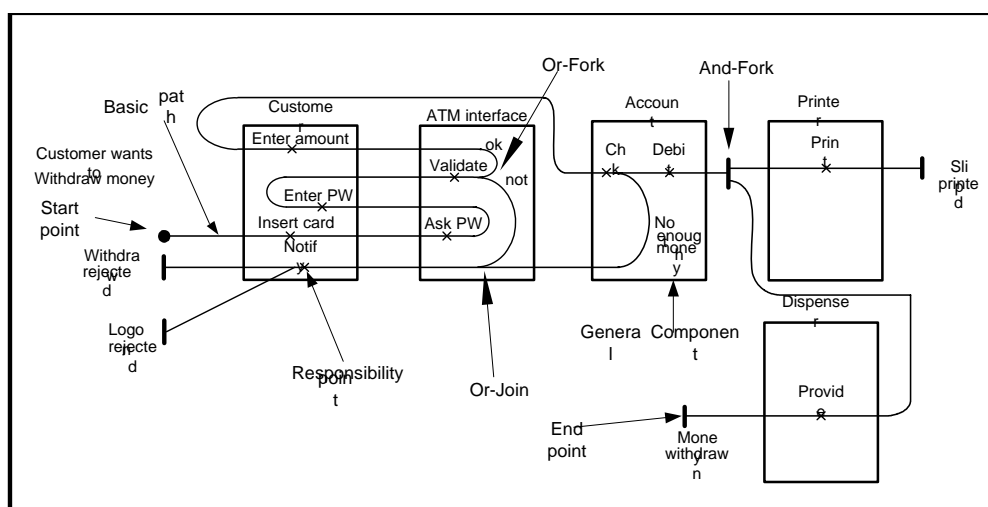


Figure A.1 UCM Scenario for Money Withdrawal

The path starts with a filled circle, which is the point where the stimuli occurs, causing movement to start progressing along the path until the end point of a path is reached. Paths trace causal sequences between start and end points. The causal sequences connect stubs and responsibilities, indicated by named points along paths ([*Enter PW*], [*Validate*], [*Enter amount*], [*Chk*], [*Debit*], [*Print*], [*Notify*] and [*Provide*]). Responsibilities are tasks or functions to be performed, or events to occur.

In this example, the activities can be allocated to abstract components (*Customer*, *ATM interface*, *Account*, *Printer* and *Dispenser*), which could be seen as objects, agents, processes, databases, or even roles or persons. UCM Paths may cross many components and components may have many paths crossing them [Elammari and Lalonde 1999].

If UCMs become too complex to be represented as a single map, it can then be broken down using a simplification of responsibilities called stubs. Stubs link to sub-maps, which are called “plug-ins”. These may be located along paths like responsibilities. Stubs are more general than stubs in three ways:

- Stubs identify the existence of sub UCMs.
- Stubs may span multiple paths (not shown).
- A stub can be static or dynamic.

Figure A.2 shows a UCM scenario for money withdrawal with stubs. Static stubs contain only one plug-in and enable hierarchical decomposition of complex maps. Dynamic stubs are usually notated with a dashed outline to differentiate them from static stubs. Dynamic stubs may symbolize numerous plug-ins whose selection can be determined at run-time. Such a selection is chosen using a selection policy, which often has described preconditions. Multiple selections of plug-ins can be done at once, either sequentially or in parallel.

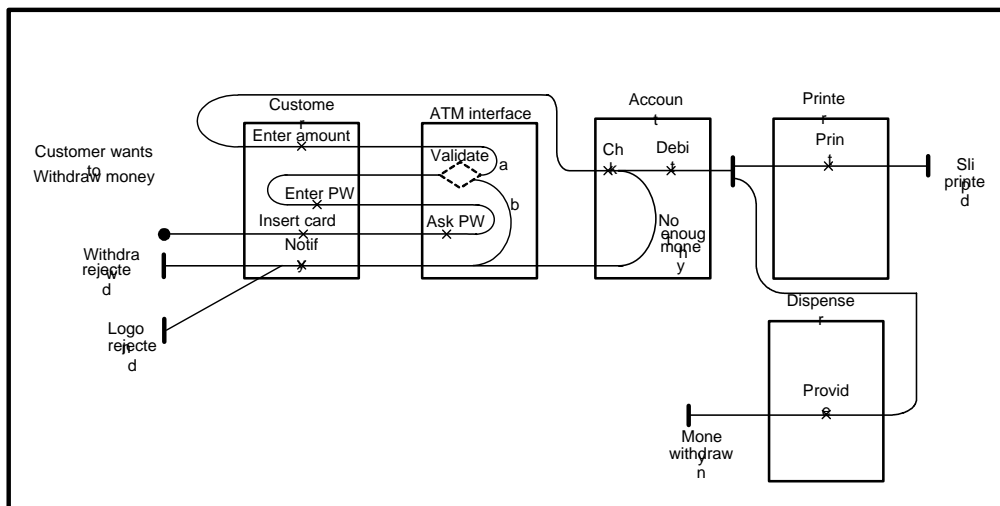


Figure A.2 UCM Scenario for Money Withdrawal with Stubs

A plug-in may involve supplementary system components not shown in the main UCM. Responsibilities and end points can have post-conditions attached, while start points may have pre-conditions. In figure A.3 the validate responsibility was substituted with a dynamic stub labeled *validate*. The *validate* stub has two outgoing ports *a* and *b*. Port *a* means authentication was accepted and port *b* means authentication was denied. There are two plug-ins connected to the validate stub: password and fingerprint.

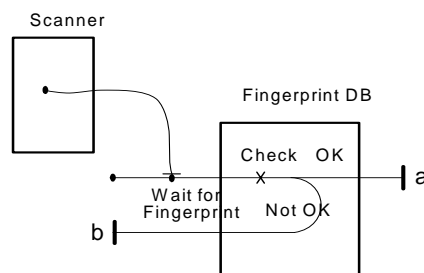


Figure A.3 Fingerprint Plug-In for the Validate Stub

Figure A.3 shows the details of the fingerprint plug-in. The fingerprint plug-in describes the behavior when the validation is performed by fingerprint. The plug-in starts with *wait for fingerprint*, which waits until the customer enters his/her fingerprint and then the path proceeds to the *check* task. It is followed by an or-fork in the path. If the entered fingerprint was found to match the stored fingerprint then the path

labeled *ok* is followed to the end point *a*. Otherwise, the path labeled *not ok* is followed to the end point *b*.

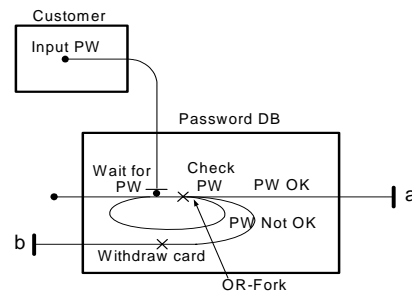


Figure A.4 Password Plug-In for the Validate Stub

The other plug-in is the password plug-in. This plug-in is utilized when the customer enters a password as an alternative for a fingerprint. In figure A.4 the password plug-in begins with *wait for PW*, which waits for the client to enter a password and then the *check PW* task is performed. The path forks into three paths after the *check PW* responsibility. The first fork (labeled *PW OK*) is ensued when the entered password matches the customer's stored password. The second fork (labeled *PW not OK*) is followed when the entered password is inaccurate. The third fork is followed if the client is permitted to retry the password after it is found to be wrong.

APPENDIX B: UML USE CASE DIAGRAMS

B.1 UML Use-Case Diagrams (UCDs)

UML use case diagrams are one of the most suitable diagrams that capture the functional requirement of the system as a whole from an external perspective. UCDs are important for visualizing, specifying, and documenting the behavior of the system. They make systems more understandable by presenting an outside view of how the system is used in context.

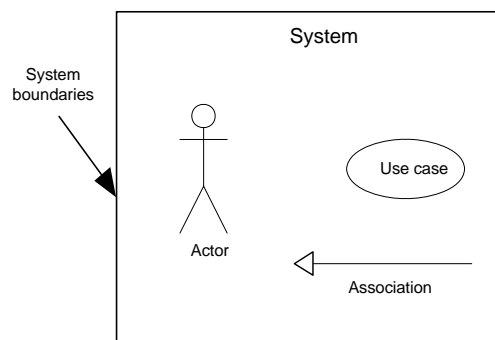


Figure B.1 Use Case Diagrams Notations

UCDs are also important for testing executable systems through forward engineering and for comprehending executable systems through reverse engineering. UCDs specify desired behaviour but they do not exhibit how that behaviour will be carried out. UCDs allow for the specification of high level user goals that the system must perform. Such goals are not necessarily events or tasks, but can be a broader obligatory functionality of the system. There is no standard template for documenting detailed use cases. There are a number of competing schemes, and individuals are encouraged to use templates that work for them or the project they are on.

UCDs commonly contain the core sections of a use case as shown in figure B.1:

1. **A use case** is drawn as a horizontal ellipse.
2. **Actors** are drawn as stick figures.
3. **Association relationships** between actors and use cases are indicated in use case diagrams by solid lines.
4. **System boundary boxes** (optional).
5. **Packages** (optional).

B.1.1 Use Case

Use cases are described a set of sequences, in which the actors (objects out with the system) and key abstractions (within the system) are represented as interactions in a sequence. These system-level functions (behaviors) are used to construct, specify, document and visualize the projected behavior of the system throughout requirements

capture and analysis. A use case is consists of a set of scenarios. Each of these is a sequence of steps that comprise of an interaction between the system and a user. Use cases assemble scenarios that accomplish specific goals of the user. For example, a user can state how an ATM system should act by affirming in use cases how users interact with the system. The client does not need to know anything about the workings of the ATM at all. This allows the user (as an end user and domain expert) communicate with the developers (who build systems that satisfy the requirements) without having to worry about the details.

Use cases represent the operative prerequisites of the system as a whole. For example, one main use case of a bank is to administer loans. A use case includes the interaction of the system and actors. An actor symbolizes a consistent set of roles that clients of use cases play when interacting with these use cases. Actors can be human beings or they can be mechanical systems. In modeling a bank, for example, administering a loan involves, along with other things, the interaction involving a customer and a loan officer.

B.1.2 Actor

UCDs allow a designer to graphically illustrate the actors within an UCD and the UCD itself. An actor is a role that a user performs in the system. It is essential here to differentiate between an actor (acknowledged also as a role) and a user.

Throughout the course of his/her/its occupation (as an actor bay be another system), a user may perform several different roles e.g. a manager, a salesperson, a support person, or a web store system. The same person may be a manager and at the same time perform the role of a salesperson. A designer is concerned more with the roles that are played, rather than the individuals.

An actor symbolizes a consistent set of roles that users of use cases perform when interacting with these use cases. An actor normally symbolizes a role that a hardware divide, a human, or still a different system, plays with a system. An example of this is someone who works in a bank may be a *loan officer*. If he/she also has an account at the bank, then the individual also plays the role of a *customer*. An actor, therefore, can symbolize an individual interacting with the system in a particular way. Actors are not actually a part of the system, although they will be used in models, as they live out with the system.

B.1.3 Association Relationships

On a UCD, associations are illustrated between actors and use cases to demonstrate that an actor performs a use case. A use case can be preformed by many actors and an actor may perform many use cases. Use cases can also be connected or associated to one other with three different links (*includes*, *generalization* and *extends*):

i. Includes

Figure B.2 shows the use of *includes* link. Both *online purchase* and *invoice purchase* include the scenarios classified by *purchase valuation*. In general, *includes* link is used to evade repetition of scenarios in multiple use-cases.

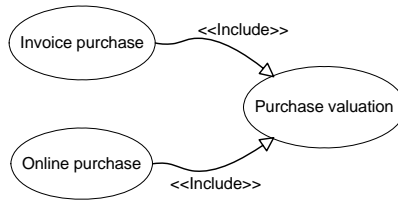


Figure B.2 Includes Relationship

ii. *Generalization*

When a use case depicts an alternative on another use case, a generalization link is used. In figure B.3, the use case limit exceeded illustrates a circumstance in which the usual scenario of online purchase is not executed.

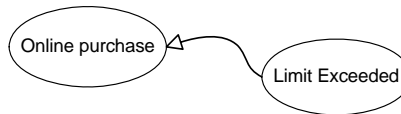


Figure B.3 Generalization Relationship

Use cases that simplify another use case should only denote an alternative, even outstanding, scenario to the use case being simplified. The general objective of the use cases should be identical.

iii. *Extends*

The extends arrow is illustrated from the use case X to the use case Y to point out that the process X is a unusual case behavior of the same type as the more general process Y. In situations where the system has a quantity of use cases (processes) that all have some mutual subtasks, this sort of link would be used. Each one, on the other hand, has different properties that prevent the designer from combining them all together in the same use case. In some cases, the variation on the behavior may need to be portrayed in a controlled form. In figure B.4, *search*, at the name extension point, is said to have been extended by the *search by name* point.

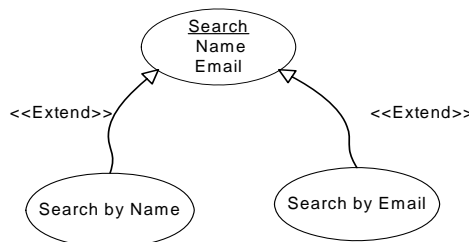


Figure B.4 Extends Relationship

B.1.4 System Boundary Boxes

A rectangle can be illustrated around the use-cases and it is called the system boundary box. The scope of the system is specified by the system boundary box. Within the box, functionality that is in scope is represented and anything outside the box is not.

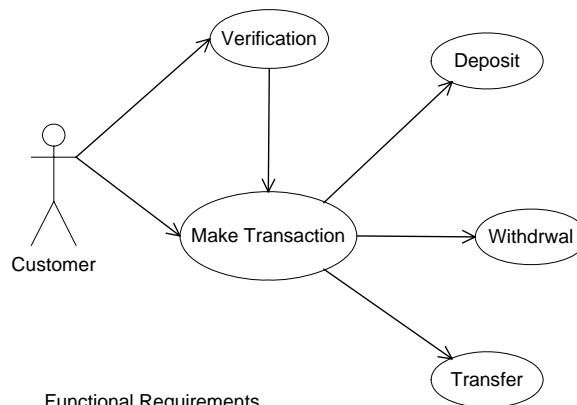
B.1.5 Useful Remarks

A simple use case model is the most important factor to keep in mind. It is often easier to ascertain the actors of the system and then flush out the use cases that they execute. UCDs can be either very simple, or very complex. Simpler diagrams are nevertheless easier to comprehend and are can portray the tasks of the system better.

Use cases can be broken down into new sub-use case diagrams. The use case *online purchase* may, for example, need additional specification as one move into the design. In each use case, sub-diagrams can be made in order to aim clarification and understanding of the tasks involved. A simple use case design aids a user's aims and expectations of the system.

B.2 UCDs by Example

Figure B. exhibits the ATM use case diagram. Functional requirements should be taken into consideration by the developer.



Functional Requirements

The ATM system shall perform customer verification before allowing the customer to make a transaction.

The ATM system shall allow the customer to make deposits, withdrawals and transfers.

Figure B.5 ATM System Use Case Diagram

Use-case name: *Verification*

Description:

- Customer inserts card into the ATM.
- System searches *stolen card* file for a record of this card.
- System gets *customer account* record from *customer accounts* file
- System displays *customer verification* window.
- System asks for *PIN*.
- Customer enters *PIN*.
- System displays *customer service* window.

Actor: Customer

Goal: To verify whether the card is stolen or not.

Preconditions:

- System is displaying *waiting for customer* window.

Postconditions:

- If any alternative course of action is taken, system displays *waiting for customer* window.

Triggering event: the card is inserted into ATM.

Extension: None.

Alternatives:

- If card is stolen, system retains card and notifies police.
- If card account number does not match that of any customer account record, system returns card.
- If customer account status is “on-hold”, system retains card.
- If customer takes to long to respond, system times out and system returns card.
- If customer fails to enter the correct PIN in three attempts, system returns card.

Use-case name: *Make Transaction*

Description:

- System waits for user to select *transaction type*.
- While transaction type does not equal *quit*.
- If transaction type is *deposit*, customer performs deposit use case.
- If transaction type is *withdrawal*, customer performs withdrawal use case.
- If transaction type is *transfer*, customer performs transfer use case.
- System returns card.
- System displays waiting for customer window.

Actor: Customer

Goal: To accomplish transactions.

Preconditions:

- Customer has performed *verification* use case - customer service window is being displayed.

Postconditions:

- None

Triggering event: the pin number is entered.

Extension: None.

Alternatives: Deposit, Withdrawal, and Transfer,

- None

APPENDIX C: UML ACTIVITY DIAGRAMS

C.1 Activity Diagrams

An activity diagram shows in a basic form a straightforward and intuitive visual of what happens in a workflow, whether there are substitute paths through the workflow, and what activities can be done together.

The Unified Modeling Language (UML) defines activity diagrams. These diagrams are a consequence of various systems that aimed to visually demonstrate workflows. Much of the basis for the definition of the activity diagram notation is found in [Martin and Odell 1996].

The workflow of the interior operation of the agent system can be modeled by activity diagrams. By reproducing the flow of control from activity to activity, they show the dynamic nature of a system. Activity diagrams replicate the actions to perform and in what sequence the agent/object will execute them. Each activity encloses an action expression that denotes the action to perform. An activity symbolizes an operation on some class in the system that causes a change in the status of the system. In many ways, UML activity diagrams are the object-oriented counterparts of data flow diagrams (DFDs) and flow charts in planned development.

Basic activity diagram notation:

- **Start state:** The filled-in circle is the root of the diagram. An initial node is not necessary although it does make it significantly easier to read the diagram.
- **End state:** The filled circle with a border is the ending point. An activity diagram can have zero or more activity final nodes.
- **Activity:** The rounded rectangles symbolize activities that occur. It represents execution of an atomic action, which represents the performance of a step within the workflow. An activity may be physical, such as *Inspect Forms*, or electronic, such as *Display Create Student Screen*.
- **Transitions/flow:** The arrows on the diagram that show what activity follows another. This type of transition can be referred to as a completion transition. It differs from a transition in that it does not require an explicit trigger event; it is triggered by the completion of the activity that the activity represents.
- **Fork:** A black bar with one flow going into it and several leaving it. This denotes the beginning of a parallel activity.
- **Join:** A black bar with several flows entering it and one leaving it. All flows going into the join must reach it before processing may continue. This denotes the end of parallel processing.
- **Decision/Branch:** A diamond with one flow entering and several leaving. It allows showing alternative flows within the workflow. The flows leaving include guard conditions.
- **Guard condition:** Text such as *[Incorrect Form]* on a flow. Once the activity has been completed, it controls which transition of a set of alternative transitions the flow follows.

- **Merge:** A diamond with several flows entering and one leaving. The implication is that one or more incoming flows must reach this point before processing can continue, based on any of guards on the outgoing flow.
- **Partition:** Is organized into three partitions, also called swim lanes, indicating who/what is performing the activities (the *Applicant*, *Registrar*, or *System*).
- **Flow final:** The circle with the X through it ⊗. This indicates that the process stops at this point.
- **Note:** A standard UML note to indicate that the merges do not require all three flows to arrive before processing can continue.

Figure C.1 shows the activity diagram with explanations of its symbols.

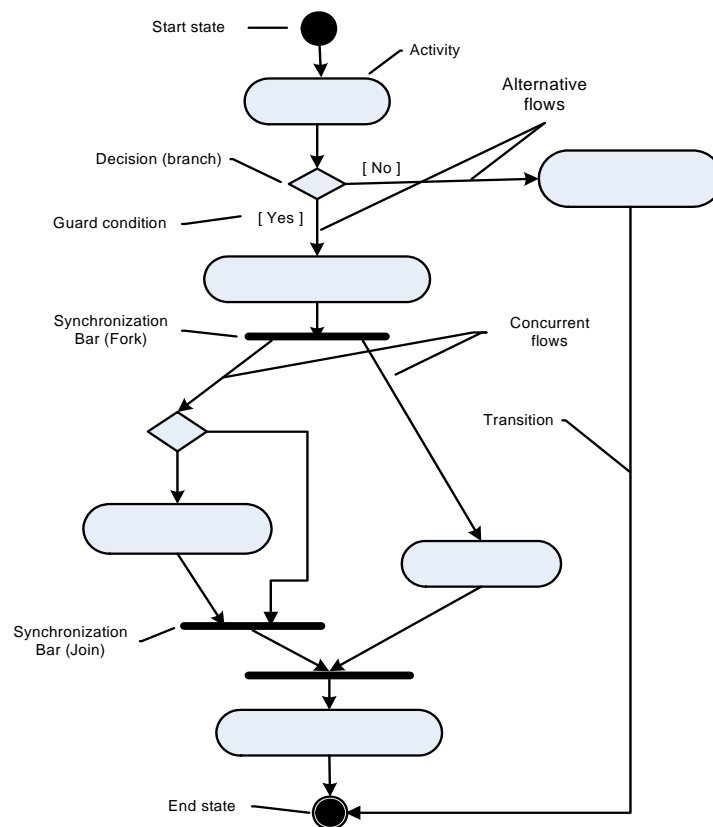


Figure C.1 Notation of Activity Diagrams

C.2 Activity Diagrams by Example

An **activity diagram** is fundamentally a complex flowchart. Activity diagrams and statechart diagrams are associated. While a statechart diagram concentrates on an object undertaking a process (or on a process as an object), an activity diagram concentrates on the flow of activities concerned in a single process. The activity diagram demonstrates how those activities are reliant on one another. The example process “Withdraw money from a bank account through an ATM” has been used.

There are 3 classes of the activity concerned: the **customer**, the **ATM**, and the **bank**. The process starts at the black start circle at and ends at the concentric white/black stop circles at the bottom. The activities are rounded rectangles.

Activity diagrams can be separated into object **swimlanes** that decide which object/agent is responsible for which activity. A separate **transition** comes out of each activity, linking it to the next activity.

Figure C.2 shows the “Withdraw money from a bank account through an ATM” example of the activity diagram with explanations of its symbols.

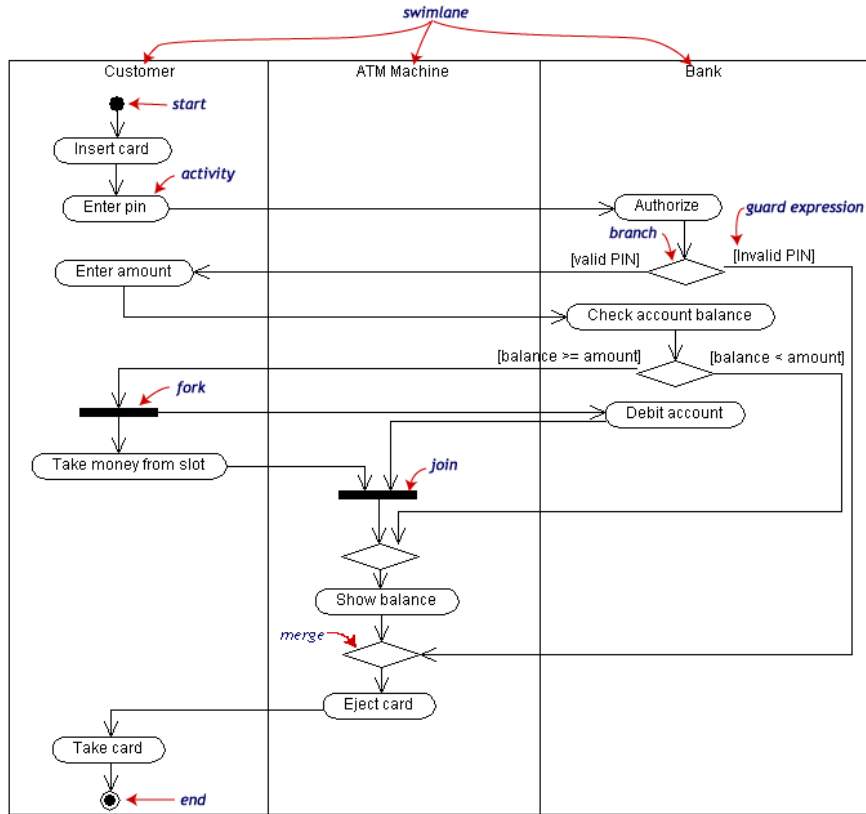


Figure C.2 Withdraw Money from a Bank Account through an ATM Activity Diagram

APPENDIX D: FIPA-ACL

D.1 FIPA-ACL

The Foundation for Intelligent Physical Agents (FIPA, www.fipa.org) began work in 1999 on a course of agent standards in which the centerpiece is an ACL. FIPA-ACL (Agent Communication Language) is a language that permits agent-to-agent communication with messages (communicative acts). Communicative acts indicate that an agent performs an action, called communicative acts. These communicative acts send messages, which are encoded. FIPA-ACL involves itself with inter-agent communication through message transferring.

- FIPA-ACL comprises three libraries as follows:
 - FIPA Communicative Act Library (CAL)
 - FIPA Content Language Library (CLL)
 - FIPA Interaction Protocol Library (IPL)
- The FIPA-ACL message structure is “filled” with concepts from the above libraries.

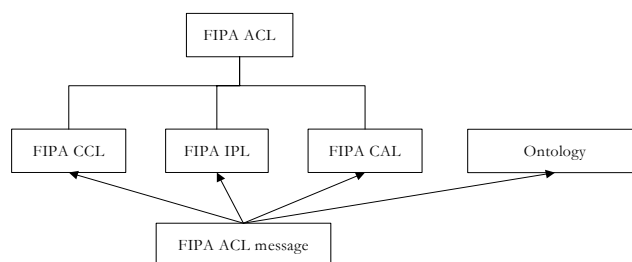


Figure D. 1 FIPA ACL Message Structure

D.1.1 FIPA Communicative Acts Library

The FIPA Communicative Acts Library is a broad catalogue of Communicative Acts that can be applicable to *any* domain. Communicative acts (CAs) are the fundamental blocks of a dialogue involving two agents. A communicative act is independent of the content and is executed by just transferring a message from one agent to another. The denotation of the communicative acts refers to mental attitude beliefs, uncertainty, and intention.

There are two categories of communicative acts:

- 1) Primitive: Those atomic actions that are not created by more than one communicative act.
- 2) Composed: Those actions created by more than one communicative act, and they can be as follows:
 - i) Making one object of another (query-if act: “I *request* you to *inform* me whether it is raining”)

- ii) Using the composition operator “;” to sequence actions. ($a;b$ means action a followed by action b)
- iii) Using the composition operator “|” to perform a particular kind of communicative act called *macro action* where a set of possible disjunctive actions ($a|b$ means a or b but not both) is available.

The following table D.1 describes the FIPA communicative acts briefly. For more information about FIPA Communicative Act Library, refer to FIPA-ACL communicative acts specification [XC00037H.pdf].

FIPA ACL Communicative Acts	
Performative	Description
<i>accept-proposal</i>	sender accepts proposal made by other agent
<i>agree</i>	sender agrees to carry out requested action
<i>cancel</i>	Follows request; indicates intention behind request is not valid any more
<i>cfp</i>	call for proposals; initiates negotiation between agents; content-parameter contains action (desired to be done by some other agent) (e.g.: "sell me car") and condition (e.g.: "price < 1000\$")
<i>confirm</i>	confirm truth of content (recipient was unsure)
<i>disconfirm</i>	confirm falsity of content (recipient was unsure)
<i>failure</i>	attempt to do requested action failed
<i>inform</i>	together with request most important performative; basic mechanism for communicating information; sender wants recipient to believe info; sender believes info itself
<i>inform-if</i>	informs other agent about truth of statement (in its content parameter) if it is true; typically content of request message (thus asking the receiver to inform me if statement is true)
<i>inform-ref</i>	informs other agent about value of expression (in its content parameter); typically content of request message (thus asking the receiver to give me value of expression)
<i>not-understood</i>	sender indicates that it recognized that an action was performed by other agent but it did not understand why it was performed. (-> error handling mechanism)
<i>propagate</i>	request to propagate a message to specified agents
<i>propose</i>	make proposal
<i>proxy</i>	same as propagate but with proxy functionality
<i>query-if</i>	direct query for the truth of a statement
<i>query-ref</i>	direct query for the value of an expression
<i>refuse</i>	reject request
<i>reject-proposal</i>	sender does not accept proposal
<i>request</i>	issue request for an action
<i>request-when</i>	issue request to do action if and when a statement is true
<i>request-whenever</i>	issue request to do action if and whenever a statement is true
<i>subscribe</i>	sender asks to be notified when statement changes

Table D. 1 Table D.1 FIPA Communicative Acts

D.1.2 FIPA Interaction Protocols Library

Protocols are patterns in which conversations between agents often fall into. A designer can make an agent complex enough, in order that the protocols arise impulsively from themselves, or the designer can state beforehand the protocol that the agents are going to pursue. A straightforward model of a protocol is given by the FIPA-request interaction protocol. This protocol permits one agent to request another to execute some action, and the receiving agent to execute the action or to reply saying that it cannot achieve it. The agents should follow these protocols in order to achieve successful conversations.

FIPA Request Interaction Protocol

The FIPA Request Interaction Protocol (IP) permits one agent to demand another to execute an action. This protocol is illustrated in Figure D.2, which is founded on extensions to UML 1.x. [Odell 2001]. This protocol is recognized by the token FIPA-request as the value of the protocol parameter of the ACL message.

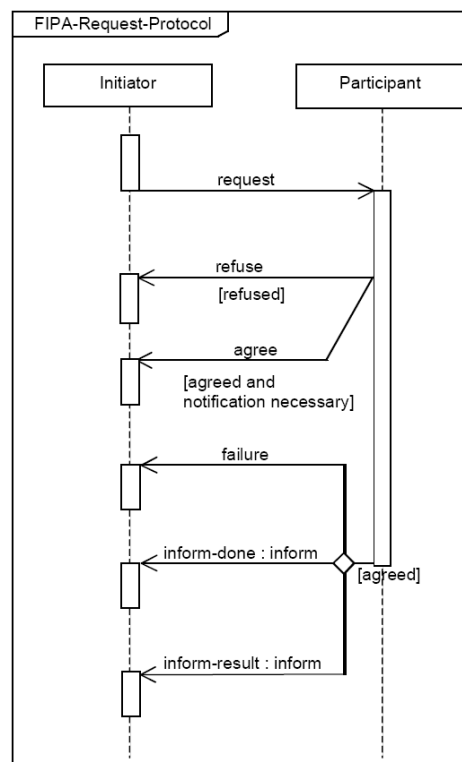


Figure D.2 FIPA Request Interaction Protocol

Explanation of the Protocol Flow

The participant administers the request and makes a decision whether to consent or decline the request. If decision is declined, then “refused” becomes true and the participant communicates a refuse. Otherwise, “agreed” then becomes true.

The participant communicates an agree if conditions signify that an explicit agreement is necessary (that is, “notification necessary” is true). The agree may be discretionary depending on situations, for example, if the requested action is very rapid and can happen before a time denoted in the reply-by parameter. As soon as the request has been agreed upon, then the participant must convey either:

- A failure if it fails in its attempt to fill the request,
- An inform-done if it successfully completes the request and only wishes to indicate that it is done, or,
- An inform-result if it wishes to indicate both that it is done and notify the initiator of the results.

Any communication using this interaction protocol is recognized by a internationally unique, non-null conversation-ID parameter, which is assigned by the initiator. All of the ACL messages must be labeled with this conversation identifier by agents concerned in the interaction. This enables each agent to administer its communication strategies and activities. It allows, for example, an agent to pinpoint individual conversations and to rationalize across historical records of conversations.

Exceptions to Protocol Flow

The receiver of a communication can notify the sender that it did not comprehend what was communicated at any point in the IP by returning a not-understood message. As such, Figure D.3 does not portray a not-understood communication as it can occur at any stage in the IP. The communication of a not-understood within an interaction protocol may terminate the complete IP and termination of the interaction may mean that any obligations made during the interaction are null and void.

At any point in the IP, the initiator of the IP may withdraw the interaction protocol by initiating the meta-protocol depicted in Figure D.3. The conversation-ID parameter of the cancel interaction is the same as the conversation-ID parameter of the interaction that the initiator aims to withdraw. The semantics of cancel should generally be understood as denoting that the initiator is no longer concerned about continuing the interaction and that it should be terminated in a method suitable to both the initiator and the participant. The participant either notifies the initiator that the interaction is complete using an inform-done or indicates the breakdown of the cancellation using a failure.

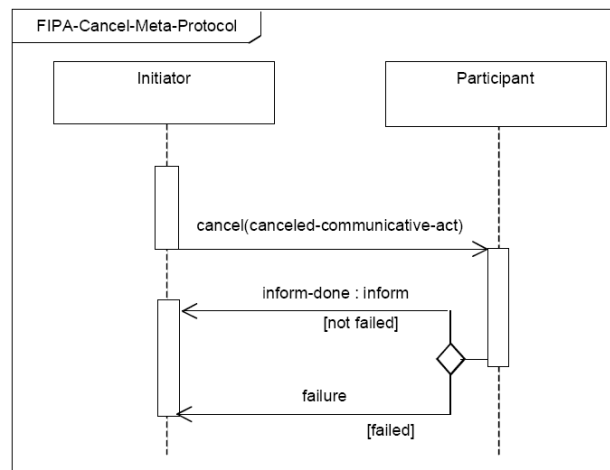


Figure D. 3 FIPA Request Interaction Protocol

This IP is a pattern for a simple interaction form. It is necessary to embellish on this pattern in order to identify all cases that might happen in a genuine agent interaction. Real-world matters such as the effects of cancelling actions, asynchrony, irregular or unexpected IP termination, nested IPs, and the like, are explicitly not tackled here.

D.1.3 Messages in FIPA ACL

It is not essential for an agent to execute every type of a message but there are some negligible necessities for an agent to be ACL compliant:

1. Agents are able to send and understand “**not-understood**” messages.
2. ACL messages must be correctly implemented according to the semantic definition.
3. ACL communicative acts must be correctly implemented according to their definitions.
4. New communicative acts should not mean the same as other pre-defined standard acts.

Agents must be able to appropriately generate a syntactically well-formed message in the carrying form that relates to the message they want to dispatch.

D.1.3.1 Message structure

A FIPA ACL-message consists of a set of one or more message parameters. The parameters are required for a successful agent communication and will differ according to the circumstances. The **performative** is the only parameter that is compulsory in all ACL messages. It is anticipated that most ACL messages will also contain a **sender**, a **receiver** and a **content** parameters. An agent can reply with the suitable **not-understood** message if it does not recognize or is not capable to administer one or more of the parameters or parameter.

When the value can be presumed by the context of the conversation, some parameters of the message may be deleted. However, FIPA does not identify any means to deal with such conditions, therefore the implementations that omit some message parameters are not assured to function with each other. The full set of FIPA ACL message parameters are clarified and is shown in Table D.2.

Parameter	Category of Parameters
performative	Denotes the type of the communicative act of the ACL message
sender	Denotes the identity of the sender of the message, that is, the name of the agent of the communicative act.
receiver	Denotes the identity of the intended recipients of the message.
reply-to	This parameter indicates that subsequent messages in this conversation thread are to be directed to the agent named in the reply-to parameter, instead of to the agent named in the sender parameter.
content	Denotes the content of the message; equivalently denotes the object of the action. The meaning of the content of any ACL message is intended to be interpreted by the receiver of the message. This is particularly relevant for instance when referring to referential expressions, whose interpretation might be different for the sender and the receiver.
language	Denotes the language in which the content parameter is expressed
encoding	Denotes the specific encoding of the content language expression
ontology	Denotes the ontology(s) used to give a meaning to the symbols in the content expression.
protocol	Denotes the interaction protocol that the sending agent is employing with this ACL message

conversation-id	Introduces an expression (a conversation identifier) which is used to identify the ongoing sequence of communicative acts that together form a conversation.
reply-with	Introduces an expression that will be used by the responding agent to identify this message.
in-reply-to	Denotes an expression that references an earlier action to which this message is a reply.
reply-by	Denotes a time and/or date expression which indicates the latest time by which the sending agent would like to receive a reply.

Table D.2 FIPA ACL Message Parameters

The following examples show some FIPA messages with some parameters:

1. Agent *i* requests *j* to open a file.

```
(request
  :sender (agent-identifier :name i)
  :receiver (set (agent-identifier :name j))
  :content
    "open \"db.txt\" for input"
  :language vb)
```

2. Agent *i* informs *j* that it accepts an offer from *j* to stream a given multimedia title to channel 19 when the customer is ready. Agent *i* will inform *j* of this fact when appropriate.

```
(accept-proposal
  :sender (agent-identifier :name i)
  :receiver (set (agent-identifier :name j))
  :in-reply-to bid089
  :content
    ((action (agent-identifier :name j)
      (stream-content movie1234 19))
     (B (agent-identifier :name j)
       (ready customer78)))
  :language FIPA-SL)
```

3. Agent *i* (a job-shop scheduler) requests *j* (a robot) to deliver a box to a certain location. *J* answers that it agrees to the request but it has low priority.

```
(request
  :sender (agent-identifier :name i)
  :receiver (set (agent-identifier :name j))
  :content
    ((action (agent-identifier :name j)
      (deliver box017 (loc 12 19))))
  :protocol fipa-request
  :language FIPA-SL
  :reply-with order567)
```

```
(agree
:sender (agent-identifier :name j)
:receiver (set (agent-identifier :name i))
:content
  ((action (agent-identifier :name j)
    (deliver box017 (loc 12 19)))
  (priority order567 low))
:in-reply-to order567
:protocol fipa-request
:language FIPA-SL)
```

4. Agent *i* confirms to agent *j* that it is, in fact, true that it is snowing today.

```
(confirm
:sender (agent-identifier :name i)
:receiver (set (agent-identifier :name j))
:content
  "weather (today, snowing)"
:language Prolog)
```

More details can be found at
<http://www.fipa.org/specs/fipa00037/XC00037H.html>

APPENDIX E: JADEX FRAMEWORK

E.1 Jadex

Based on the concept of agents with mental states, intelligent agents are a modeling example. The Jadex reasoning engine follows the Belief Desire Intention (BDI) model. It assists easy intelligent agent production with sound software engineering fundamentals. It allows for programming intelligent software agents in XML and Java and can be installed on different varieties of middleware such as JADE.

Several different constituent elements have to be built in order to allow the creation of intelligent agents. It is essential to supply an agent architecture that takes into account agent-internal, agent society and artificial intelligence (AI) theories. It is a asset of agent research that fascinating research results in various isolated areas are present, but that these outcomes are not incorporated into super ordinated architectures. As a result, until now no standards can address the creation of intelligent agents in all features.

The Jadex project assists these properties with an open research map that summarizes the research areas of concern and the actual work in development in these areas. Everyone is invited to contribute his/her ideas and practical improvements as in the spirit of an open-source project.

E.2 Features

The following sections highlight some of the present characteristics of Jadex. In summary, Jadex is a Java based, allows the development of goal-oriented agents following the BDI model, and is a FIPA compliant agent environment. Jadex offers a framework and a set of improvement tools to simplify the construction and testing of agents.

E.2.1 Java Based

Without forgoing the expressional ability of the agent paradigm, the Jadex project aims to make the growth of agent-based systems as easy as possible. To promote a smooth conversion from traditional distributed systems to the development of (multi-) agent systems, ingrained object-oriented concepts and technologies should be utilized wherever possible. It is feasible to create agent systems without having to study a new programming language by using Jadex. It is designed to assist the implementation of agents in the extensive Java programming language, therefore permitting the reuse of a immense amount of on hand tools and libraries.

E.2.2 FIPA Compliant

The opportune availability of standards to assure interoperability between growing products is one of the main success factors of a new technology. In order to assist the interoperability of independently developed (multi-) agent systems, the Foundation for Intelligent Physical Agents (FIPA) issued a set of specifications, which are generally called “the FIPA standard”. As shown in figure E.1, the FIPA standard indicates an agent platform architecture, which classifies services such as agent management and a directory facilitator. This architecture allows agents to correspond using a common agent communication language.

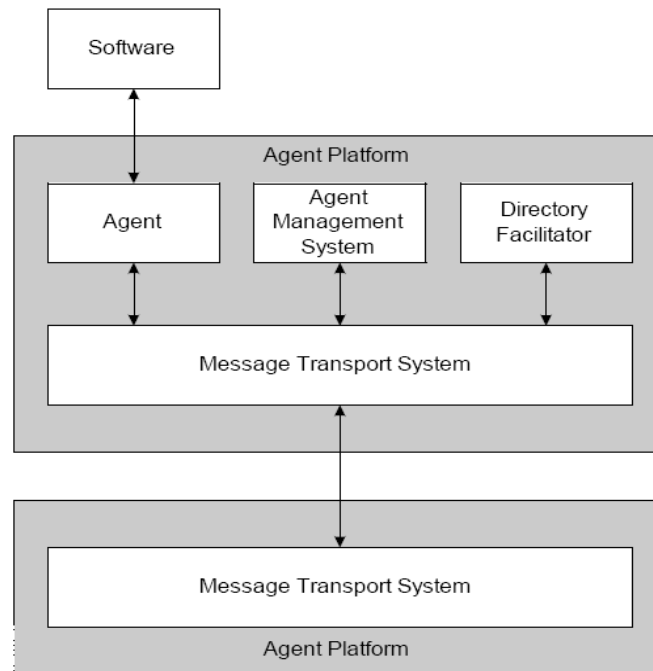


Figure E.1 FIPA Agent Management

Jadex is based on the JADE Agent Framework, an open source development by the Telecom Italia Lab, in order to attain FIPA-compliance. JADE offer the platform architecture and the central services and message transport mechanisms as commanded by the FIPA specifications. Further benefits of using JADE occur from its stability, characteristics such as agent deployment and debugging tools, and its considerable and active user base. Figure E.1 shows FIPA agent management.

E.2.3 Goal-Oriented Agents

The agent notion is regarded as a significant software development paradigm and is highly suited to address the complexity of today’s significant software systems. It permits the presentation of a system as being organized of autonomous cooperating entities, which act in a rational manner and chase their own goals. The internal state and determination process of agents is consequently modeled in an instinctive approach following the concept of mental attitudes. Goal orientation means that, instead of directly demanding the agents to execute certain actions, the developer can classify goals that are conceptual for the agents, in this manner supplying a certain amount of flexibility on how to accomplish the goals.

The BDI Model, based on the mental attitudes belief, desire, and intention, was first presented as a philosophical model for modeling reasonable (human) agents, but was later taken on and changed into an implementation model for software agents, which was the foundation on the notion of beliefs, goals, and plans. This model is integrated into JADE agents by Jadex, through introducing beliefs, goals, and plans as unparalleled objects that can be produced and influenced inside the agent. Agents have beliefs in Jadex, which can be any sort of Java object and are accumulate in a belief base. Goals are implicit or explicit explanations of states to be realized. To accomplish its goals the agent carries out plans, which have procedural formula coded in Java.

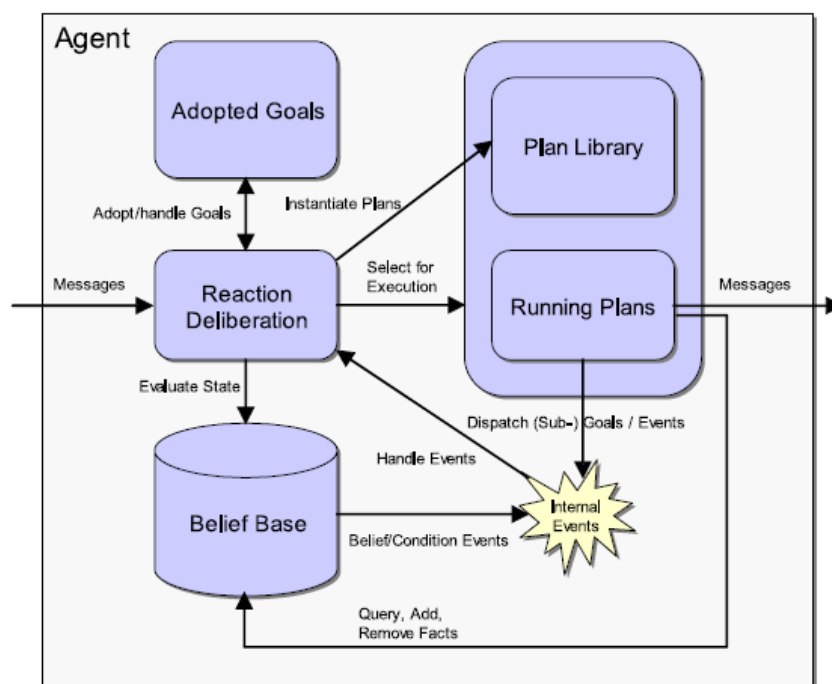


Figure E.2 FIPA Agent Management.

E.2.4 Framework

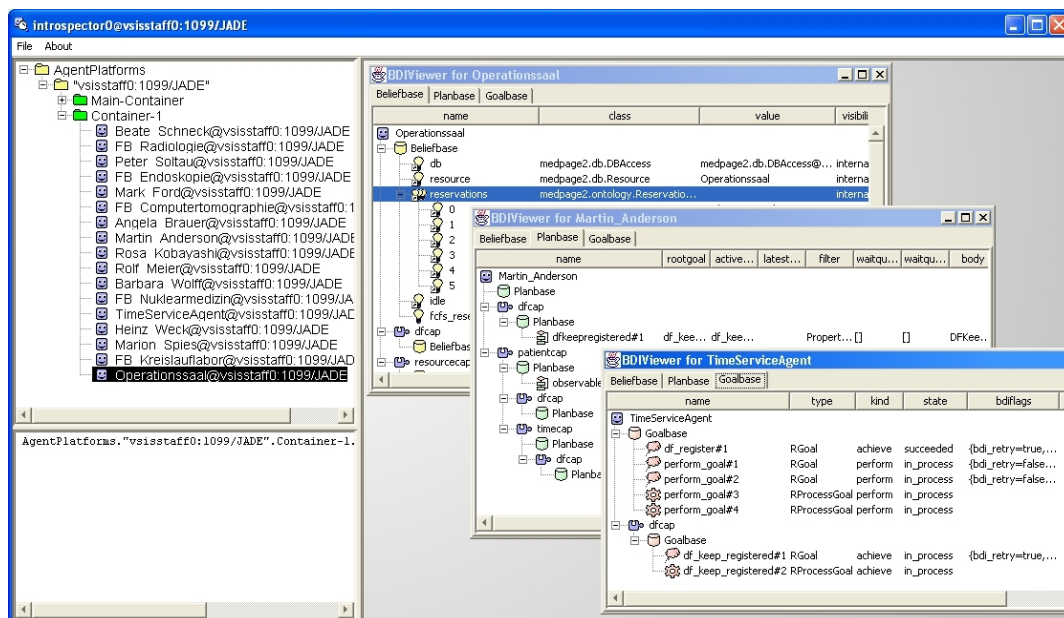
The Jadex framework is composed of API, an execution model, and predefined reusable generic functionality. The API offers admission to the Jadex theories when programming plans. Plans are plain Java classes, developing a specific abstract class, which provides practical technique e.g. for transferring messages, transmitting sub goals or waiting for events. By using the API of the belief base, plans are able to read and change the beliefs of the agent. A special feature of Jadex is that in addition to recovering saved facts, an intuitive OQL-like query language allows a formulation of arbitrary complex expressions using the objects enclosed in the belief base.

In addition to the plans coded in Java, the developer supplies an XML based Agent Definition File (ADF), which identifies the initial beliefs, goals, and plans of an

agent. The Jadex runtime engine interprets this file to instantiate an agent model, and performs the agent by keeping track of its goals while constantly selecting and executing plan steps, founded on internal events and messages from other agents. Jadex is provided with some predefined functionality e.g. to admission a directory facilitator service. The functionality, coded in unconnected plans, is written in reusable agent modules called capabilities, portrayed in a format similar to the ADF, and can be effortlessly plugged into existing agents.

E.2.5 Development Tools

Obtainable tool support is a significant quality aspect of any development environment. Jadex is built on top of JADE and there are, as a result, many readily available tools that can be used with Jadex. This is not only true for the tools included in JADE, such as the Sniffer or the DummyAgent, but also regards third party tools like the beangenerator plug-in for the ontology design tool Protégé.



Alternatively, the new concepts presented by Jadex have to be supported as well. Therefore, tools have been created to aid the developer to deal with these features e.g. related to the BDI model. The BDI Viewer tool allows the presentation of the internal state of a Jadex agent, that is, its current beliefs, goals, and plans (see picture). The Jadex Introspector is comparable to the JADE Introspector, allowing observing and manipulation of the execution of an agent, by observing and influencing how incoming events are handled. For debugging reasons, the Introspector also allows to put an agent into single-step mode (shown in the screenshot). As well as the Jadex specific tools, a Logger Agent is provided, which allows the compilation and presentation of log messages from JADE and Jadex agents, following the Java Logging API.

REFERENCES

- Abdelaziz, T., Elammari, M. and Unland, R., “*A Framework for the Evaluation of Agent-oriented Methodologies*”, In: 4th International Conference on Innovations in Information Technology. Dubai, UAE, 2007.
- Adams, F. and Campbell, K., “*Modality and Abstract Concepts*”, Behavioral and Brain Sciences, Pages 22, 610, 1999.
- AgenTool, 2000. : In <http://www.cis.ksu.edu/~sdeloach/ai/agentool.htm>.
- Agtivity. :In <http://www.agtivity.com>.
- Alonso, F., Frutos, S., Martínez, L., and Montes, C., “*SONIA: A Methodology for Natural Agent Development*”, ESAW, 2004.
- Amyot, D., “*About Use Case Maps*”, 2006. : In <http://www.usecasemaps.org/aboutu-cms.shtml>.
- Amyot, D., “*Use Case Maps and UML for Complex Software-Driven Systems*”, Technical Report, 1999. : In <http://www.usecasemaps.org>.
- Amyot¹, D. and Mussbacher, G., “*Bridging the Requirements/Design Gap in Dynamic Systems with Use Case Maps (UCMs)*”, ICSE 2001,Pages 743-744, Toronto, Canada, 2001.
- Amyot², D. and Mussbacher, G., “*Use Case Maps Bridging The Gap Between Requirements And Design*”, RE'01, Toronto, Canada, 2001.
- Amyot, D. and Mussbacher, G., “*On the Extension of UML with Use case Maps Concepts*”, UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, 2000.
- Arazy, O. and Woo, C., “*Analysis and design of agent-oriented information systems*”, The Knowledge Engineering Review, 17(2), 2002.
- Arenas, A. E. García-Ojeda, J. C. and Pérez-Alcázar, J. J., “*On Combining Organisational Modelling and Graphical Languages for the Development of Multiagent Systems*”, Integrated Computer-Aided Engineering, 11 (2) Pages 151-163 Hojjat Adeli (Eds), IOS Press, 2004.
- Armstrong, D., “*A Materialist Theory of Mind*”, London: Routledge, 1968.
- Avison, D. E. and Fitzgerald, G., “*Where now for development methodologies?*”, Communications of the ACM 46 (1) Pages 79-82., 2003.
- Avison, D. and Fitzgerald, G., “*Information Systems Development: Methodologies, Techniques and Tools*”, McGraw-Hill, New York, 2nd edition, 1995.
- Ayer, A.J., “*Logical Positivism*”, New York: The Free Press. 1959.

- Barsalou, L., “*Perceptual Symbol Systems*”, Behavioral and Brain Sciences, 22, Pages 577-609, 1999.
- Bauer, B. and Odell, J., “*UML 2.0 and agents: how to build agent-based systems with the new UML standard*”, Journal of Engineering Applications of Artificial Intelligence Vol. 18, Issue 2, 2005.
- Belina, F., Hofgreffe, D. and Sarma, A., “*SDL with Applications from Protocol Specification*”, Prentice Hall Int., Hertfordshire, UK, 1991.
- Berard E.V., “*A comparison of object-oriented methodologies*”, Technical report, Object agency Inc., 1995.
- Bobkowska, A. E., “*Framework for methodologies of visual modeling language evaluation*”, Proceedings of the symposia on Metainformatics, ACM Press 2005.
- Booch, G., Rumbaugh, J. and Jacobson, I., “*The Unified Modeling Language User Guide*”, Addison Wesley, 1998.
- Bordeleau, F. and Buhr, R.J.A., “*The UCM-ROOM Design Method: from Use Case Maps to Communicating State Machines*”, In: Conference on the Engineering of Computer-Based Systems, Monterey, USA, 1997.
- Bordeleau, F., “*A Systematic and Traceable Progression from Scenario Models to Communicating Hierarchical Finite State Machines*”, Ph.D. thesis, SCS, Carleton University, Ottawa, Canada, 1999.
- Bordini, R. H., Dastani, M., Dix, J. and El Fallah Seghrouchni A., editors, “*Multi-Agent Programming: Languages, Platforms and Applications*”, Number 15 in Multiagent Systems, Artificial Societies, and Simulated Organizations. Springer, 2005.
- Bordini, R. H., Hubner, J. F., et al, “*Jason: A Java-based agentSpeak interpreter used with saci for multi-agent distribution over the net*”, manual, first release edition, 2004. : In <http://jason.sourceforge.net/>.
- Bordini, R. H., Hübner, J. F., et al. “*Jason*”, manual, release 0.7 edition, Aug. 2005. : In <http://jason.sf.net/>.
- Bratman, M.E., “*Intentions, Plans, and Practical Reason*”, Harvard University Press, Cambridge, 1987-1999.
- Braubach, L., Pokahr, A. and Lamersdorf W., “*Jadex: A Short Overview*”, in: Main Conference Net.ObjectDays, Erfurt, Germany, 2004.
- Braubach, L., Pokahr, A., Moldt, D. and Lamersdorf W., “*Goal representation for BDI agent systems*”, In R. Bordini, M. Dastani, J. Dix . and A. El Fallah Seghrouchni, editors, Programming Multi-Agent Systems, second Int. Workshop (ProMAS’04), vol. 3346 of LNAI, Pages 44–65. Springer Verlag, 2005.
- Brazier, F., Jonker, C. and Treur, J., “*Principles of compositional multi-agent system development*”,

In Proceedings of Conference on Information Technology and Knowledge Systems, Pages 347–360. Austrian Computer Society, 1998.

Brazier, F. M. T., Dunin-Keplicz, B., Jennings, N. and Treur, I., “*Desire: Modeling Multi-Agent Systems in a Compositional Formal Framework*”, Int. Journal of Cooperative Information Systems, Vol. 6. Special Issue on Formal Methods in Cooperative Information Systems: Multi-agent Systems, 1997.

Bresciani, P., Giorgini, P., Giunchiglia, F., Mylopoulos, J. and Perini, A., “*Troops: An agent-oriented software development methodology*”, Technical Report DIT-02-0015, University of Trento, Department of Information and Communication Technology, 2002.

Bresciani, P., Giorgini, P., Giunchiglia, F., Mylopoulos, J. and Perini, A., “*Tropos: an agent-oriented software development methodology*”, J. Autonomous and Multi-Agents (in press), 2003.

Buhr, R.J.A., “*Use Case Maps as Architectural Entities for Complex Systems*”, IEEE Transactions on Software Engineering. Vol. 24, No. 12, Pages 1131-1155, 1998.

Buhr, R.J.A. and Casselman, R.S., “*Use Case Maps for Object-Oriented Systems*”, Prentice-Hall, USA, 1995.

Burmeister, B., “*Models and Methodology for Agent-Oriented Analysis and Design*”, In Working Notes of the KI'96 Workshop on Agent-Oriented Programming and Distributed Systems, Saarbrilcken, Germany, 1996.

Burrafato, P. and Cossentino, M., “*Designing a multi-agent solution for a bookstore with the PASSI methodology*”, AOIS@CAiSE, 2002.

Busetta, P., Rönquist, R., Hodgson, A. and Lucas, A., “*JACK Intelligent Agents Componentets for Intelligent Agents in Java*”, updated from AgentLink Newsletter, <http://www.agent-software.com.au/>, 1999.

Bush, G., Cranefield, S. and Purvis, M., “*The Styx Agent Methodology*”, The Information Science Discussion Paper Series, Number 2001/02, University of Otago, New Zealand, 2001.

Cabri, G., Ferrari, L. and Leonardo, L., “*Rethinking agent roles: extending the role definition in the BRAIN framework*”, Systems, Man and Cybernetics, IEEE International Conference on Vol. 6, Pages: 5455 - 5460, vol.6, 10-13 Oct. 2004.

Caire, G., Leal, F., Chainho, P., Evans, R., Garrijo F., Gomez, J., Pavon J., Kearney, P., Stark, J. and Massonet, P., “*Agent oriented analysis using MESSAGE/UML*”. In Second International Workshop on Agent-Oriented Software Engineering (AOSE-2001), Pages 101-108, 2001.

Caire, G., Leal, F., Chainho, P., Evans, R., Jorge, F. G., Pavon, J., G., Kearney, P., Stark, J. and Massonet, P., “*Methodology for agent-oriented software engineering*”, Technical Information Final version, European Institute for Research and Strategic Studies in Telecommunications (EURESCOM), 2001.

Calisti, M., Funk, P., Biellman, S. and Bugnon, T., “*A multi-agent system for organ transplant management*”, In A. Moreno and J. Nealon, editors, *Applications of Software Agent Technology in the HealthCare Domain*, Pages 199–212. Birkhäuser Verlag, 2004.

Castro, J., Candida, R., Castor, A. and Mylopoulos, J., “*Requirements Traceability in Agent Oriented Software Engineering*”, Book chapter In *Software Engineering for Large-Scale Multi-Agent Systems: Research Issues and Practical Applications*, LNCS 2603, Springer Verlag. 2003.

Castro, J., Kolp, M. and Mylopoulos, J., “*Towards requirements-driven information systems engineering*”, the TROPOS project. *Information Systems*, 27: Pages 365-389, 2002.

Cernuzzi, L. and Rossi, G. “*On the evaluation of agent oriented modeling methods*”, In *Proceedings of Agent Oriented Methodology Workshop*, Seattle, 2002.

Castro, J., Kolp, M. and Mylopoulos, J., “*A requirements-driven development methodology*”, In *Proceedings of the 13th International Conference on Advanced Information Systems Engineering (CAiSE'01)*, Interlaken, Switzerland, 2001.

Clark, K. and McCabe, F., “*Go! A multiparadigm programming language for implementing multi-threaded agents*”, *Annals of Mathematics and Artificial Intelligence*, 41(2–4): Pages 171–206, 2004.

Cohen, P.R. and Levesque H.J., “*Intention is choice with commitment*”, *Artificial Intelligence*, 1990.

Collinot, A., Carle, P. and Zeghal, K., “*Cassiopeia: A Method for Designing Computational Organizations*”, *Proc. of the First Int. Workshop on Decentralized Intelligent Multi-Agent Systems*. Krakow, Poland, Pages 124-131, 1995.

Costantini, S. and Tocchio, A., “*A logic programming language for multi-agent systems*”, In *Proceedings of the 8th European Conference on Logics in Artificial Intelligence (JELIA'02)*, vol. 2424 of LNAI, Pages 1–13, Springer, 2002.

Constantine, L. and Lockwood, L., “*Structure and style in use cases for user interface design, Object modeling and user interface design: designing interactive systems*”, Addison-Wesley Longman Publishing Co., Inc., Boston, 2001.

Cossentino, M. and Potts, C., “*PASSI: A process for specifying and implementing multi-agent systems using UML*”, (2002). : In http://www.cc.gatech.edu/classes/AY2002/cs6300_fall/ICSE.pdf.

Dam, K. H. and Winikoff, M., “*Comparing Agent-Oriented Methodologies*”, the International Bi -Conference Workshop on Agent-Oriented Information Systems (AOIS), 2003.

Dam, K. H., and Winikoff, M., “*Survey on Agent-Oriented Methodologies*”, 2002. : In <http://yallara.cs.rmit.edu.au/~kdam/Questionnaire/Questionnaire.html>.

Dastani M., Hulstijn, J., Dignum, F. and Meyer, J., “*Issues in Multi-agent System*

Development’, AAMAS, 922-929, 2004.

Dastani M., “*AgentLink-III Technical Forum Group, Programming Multi-agent System, PROMAS*”, Report, 2004.

D’Inverno, M. and Luck, M., “*Understanding agent systems?*”, agent landscape, Pages 3, 2004. : In <http://www.springer.com/978-3-540-40700-3>.

D’Inverno, M., Kinny, D., Luck, M. and Wooldridge, M., “*A Formal Specification of dMARS*”, Tech. Rep. 72, Australian Artificial Intelligence Institute, Melbourne, Australia, 1997.

DeLoach, S. A., “*The MaSE Methodology*”, In Methodologies and Software Engineering for Agent System, The Agent-Oriented Software Engineering Handbook Series: Multi-agent Systems, Artificial Societies, and Simulated Organizations, Vol. 11. Bergenti, Federico; Gleizes, Marie-Pierre; Zambonelli, Franco (Eds.) Kluwer Academic Publishing (available via Springer), 2004.

DeLoach, S. A., Wood, M. F. and Sparkman, C. H., “*Multi-agent systems engineering*”, International Journal of Software Engineering and Knowledge Engineering, 11(3): Pages 231-258, 2001.

Dittrich K. R., Gatzju S., Geppert A., “*The Active Database Management System Manifesto A Rulebase of ADBMS Features*”, Lecture Notes in Computer Science 985 ISBN 3-540-60365-4, Springer, pages 3-20, 1995.

Edmunson, S. A., Botterbusch, R. D. and Bigelow, T. A., “*Application of System Modeling to the Development of Complex Systems*”, In Proceedings of the Digital Avionics Systems Conference, IEEE/AIAA 11th Vol, Issue, 5-8 : Pages 138 - 142, 1992.

Elammari, M. and Lalonde, W., “*An Agent-Oriented Methodology: High-Level and Intermediate Models*”, HLLIM, Proceedings of AOIS-1999, Heidelberg 1999.

Eric S. K. Y. and Cysneiros, L., “*Agent-Oriented Methodologies - Towards A Challenge Exemplar*”, Proceedings of the Fourth International Bi-Conference Workshop on Agent-Oriented Information Systems AOIS@CAiSE, Toronto, Ontario, Canada, 2002.

EU-Rent, EU-Corporation. : In <http://www.businessrulesgroup.org/egsbrg.shtml>.

FIPA.: In <http://www.fipa.org/>.

FIPA ACL. : In <http://www.fipa.org/repository/aclspecs.html>.

FIPA-OS. : In <http://jmvidal.ece.sc.edu/talks/fipa-os/>.

Fisher, M., “*Representing and executing agent-based systems*”, In Wooldridge, M. and Jennings, N. editors, Intelligent Agents, Lecture Notes in Artificial Intelligence, Springer-Verlag, Vol. 890, Pages 307-323., Berlin, 1995.

Franklin, S. and Gasser, A., “*Is it an Agent, or just a Program?: A Taxonomy for Autonomous*

Agents”, Proc. of ATAL, 1996.

Genesereth, M. R. and Nilsson, N., “*Logical Foundations of Artificial Intelligence*”, Morgan Kaufmann Publishers: San Mateo, CA., 1987.

Gervais, M., “*ODAC: An Agent-Oriented Methodology Based on ODP*”, Journal of Autonomous Agents and Multi-Agent Systems, Vol. 7(3), Pages 199-228, 2003.

Giunchiglia, F., Mylopoulos, J. and Perini, A., “*The Tropos software development methodology: Processes, Models and Diagrams*”. In Third International Workshop on Agent-Oriented Software Engineering, 2002.

Glaser, N., “*The CoMoMAS Methodology and Environment for Multi-Agent System Development*”, In Multi-Agent Systems - Methodologies and Applications, LNAI 1286. Springer-Verlag, Pages 1-16, Berlin,1997.

Georgeff, M. P., Pell, B., Pollack, M. E., Tambe, M. and Wooldridge M., “*The DBI belief-desire-intention model of agency*”, In Jörg P. Möuller, Munindar P. Singh, and Anand S. Rao, editors, ATAL, vol. 1555 of Lecture Notes in Computer Science, Pages 1–10. Springer, 1998.

Goodwin, R., “*Formalizing Properties of Agents*”, Journal of Logic and Computation, 5(6): Pages 763-781, 1995.

Graham, I., Henderson-Sellers, B., Younessi, H., “*The OPEN Process Specification*”, Addison-Wesley, 1997.

Hay D. and Healy K. A., “*Defining Business Rules - What Are They Really?*”, Final Report, revision 1.3, 2000. : In http://businessrulesgroup.org/first_paper/br01c0.htm.

Henderson-Sellers, B. and Giorgini, P., “*Agent-oriented Methodologies*”, Idea Group, 2005.

Hindriks, K.V., de Boer, F.S., van der Hoek W., and Meyer, J. J., “*Agent programming in 3APL*”, Autonomous Agents and Multi-Agent Systems, 2(4): 357-401, 1999.

Hong, S., Van den Goor, G. and Brinkkemper, S., “*A formal approach to the comparison of object-oriented analysis and design methodologies*”, In The Twenty-Sixth Annual Hawaii International conference on System Sciences, Pages 689-699, Hawaii, 1993.

Huber M. J., “*JAM: A BDI-theoretic Mobile Agent Architecture*”, In AgentLink News, Issue 5, pages 2-5, 2000.

IBM agent. : In <http://activist.gpl.ibm.com:81/WhitePaper/ptc2.htm>.

Iglesias C., Garrijo, M. and Gonzalez, J., “*A survey of agent-oriented methodologies*”, In proceedings of the 5th International Workshop on Intelligent Agents V: Agent Theories, Architectures, and Languages (ATAL-98), vol. 1555, Springer-Verlag, Pages 317-330, Heidelberg, Germany, 1999.

Iglesias, C.A., Garrijo, M., Gonzalez, J.C. and Velasco, J. R., “*Analysis and Design of Multi-agent Systems using MAS-CommonKADS*”, In Intelligent Agents IV: Agent Theories,

Architectures, and Languages (ATAL97), LNAI 1365. Springer-Verlag, Pages 313-326, Berlin,1999.

Iivari, J., Hirschheim, R. and Klein, H., “*Beyond Methodologies: Keeping up with Information Systems Development Approaches through Dynamic Classification*”, HICSS, 1999.

Intelligent Agent Factory, 2000. : In <http://www.bitpix.com>.

JADE. Java Agent Development Framework, 1999. : In <http://jade.cselt.it>.

JATLite, 2000. : In <http://java.stanford.edu/java-agent/html/>.

Jayaratna, N., “*Understanding and evaluating methodologies, NISAD: A systematic framework*”, Maidenhead, UK: McGraw-Hill, 1994.

Jennings, N.R., Sycara, K. and Wooldridge, M., “*A Roadmap of Agent Research and Development*”, In: Autonomous Agents and Multi-Agent Systems Journal, Jennings, Kluwer Academic, Vol. 1, Issue 1, Pages 7-38, Publishers, Boston, 1998.

Jennings, N. R. and Wooldridge, M., “*Agent-Oriented Software Engineering*”, in proceedings of the 9th European Workshop on Modelling Autonomous Agents in a Multi-Agent World: Multi-Agent System Engineering (MAAMAW-99), vol. 1647, Springer-Verlag: Heidelberg, Germany, 1999.

Juan, T., Pierce, A. and Sterling, L., “*Roadmap: Extending the gaia methodology for complex open systems*”, In Proceedings of the 1st ACM Joint Conference on Autonomous Agents and Multi-Agent Systems (Bologna, Italy), ACM, New York, Pages 3–10, 2002.

Juan, T., Sterling, L. and Winikoff, M., “*Assembling Agent-Oriented Software Engineering Methodologies from Features*”, in the Proceedings of the Third International Workshop on Agent-Oriented Software Engineering, at AAMAS’02, Bologna, Italy, 2002.

Kendall E., “*Agent Roles and Role Models: New Abstractions for Multi-agent System Analysis and Design*”, Proceedings of the International Workshop on Intelligent Agents in Information and Process Management, Bremen, Germany, 1998.

Kendall, E. A., Malkoun, M. T. and Jiang, C. H., “*A Methodology for Developing Agent Based Systems*”, In Distributed Artificial Intelligence Architecture and Modeling, LNAI 1087. Springer-Verlag, Pages 85-99, Germany,1996.

Kinny, D., Georgeff, M. and Rao, A., “*A methodology and modelling technique for systems of BDI agents*”, In Proceedings of the Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World, LNAI Vol. 1038, Springer-Verlag, Pages 56-71, Berlin, 1996.

Kruchten, P., “*The Rational Unified Process: An Introduction*”, Addison-Wesley Pub Co, 2nd edition, 2000.

Krupansky, J. W., “*Foundations of Software Agent Technology*”, Agtivity: Advancing the Science of Software Agent Technology, <http://www.agtivity.com>. 2008.

- LEAP, 2000. : In <http://leap.crm-paris.com/>.
- Leite, J. A., “*Evolving Knowledge Bases*”, vol. 81 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2003.
- Leite, J. A., Alferes, J. J. and Pereira, L. M., “*MINERVA- a dynamic logic programming agent architecture*”, *Intelligent Agents VIII Agent Theories, Architectures, and Languages*, vol. 2333 of *LNAI*, Pages 141– 157. Springer, 2002.
- Lind, I., “*Iterative Software Engineering for Multiagent Systems: The MASSIVE method*”, *LNCS-1994*. Springer-Verlag, 2001.
- Luck, M., McBurney, P. and Preist, C., “*Agent Technology: Enabling Next Generation Computing (A Roadmap for Agent Based Computing)*”, *AgentLink*, 2003.
- Luck, M., McBurney, P., Shehory, O. and Willmott, S., “*Agent Technology: Computing as Interaction (A Roadmap for Agent Based Computing)*”, *AgentLink*, 2005.
- Maes, P., “*Artificial Life Meets Entertainment: Lifelike Autonomous Agents*”, *Commun. ACM* 38(11): Pages 108-114, 1995.
- Maes, P., “*The Agent Network Architecture*”, *SIGART Bulletin*, 2(4): Pages 115-120, 1991.
- Luck, M., Griffiths, N. and d’Inverno M., “*From agent theory to agent construction: A case study*”, In J. P. Muller, M. Wooldridge, and N. R. Jennings, editors, *Intelligent Agents III LNAI Vol. 1193*, Springer-Verlag, Pages 49-64, Berlin, Germany, 1997.
- MADKIT., “*Multi-Agent Development KIT*”, 1999. : In <http://www.madkit.org/>.
- Morrow C., “*Misconceptions Scientists Often Have about the National Science Education Standards*”, White paper available from the Space Science Institute, 2000.
- Moulin, B. and Cloutier, L., “*Collaborative Work Based on Multi-Agent Architectures: A Methodological Perspective*”, In *Soft Computing: Fuzzy Logic, Neural Networks and Distributed Artificial Intelligence*. Prentice-Hall, N.J., Pages 261-296, USA 1994.
- Mylopoulos, J., Kolp, J. and Castro, J., “*UML for agent-oriented software development: the Tropos proposal*”, in «UML»2001 – *The Unified Modeling Language*, *LNCS Vol. 2185*, Springer-Verlag, Pages 422-441, 2001.
- Odell J., “*Objects and agents compared*”, *Journal of Object Technology*, 1(1): Pages 41-53, 2002.
- Odell, J., et-al., “*Agent UML*”, 2001. : In <http://www.auml.org/>.
- Odell, J., Parunak, H., Fleischer, V. and Breuckner, S., “*Modeling Agents and their Environment*”, *Agent-Oriented Software Engineering (AOSE) III*, Springer, Berlin, 2002.
- Odell, J., Van Dyke P.H. and Bauer, B., “*Representing Agent Interaction Protocols in UML*”, In,

Agent-Oriented Software Engineering, Springer, Pages 121-140, Berlin, 2001.

Odell, J., Parunak, H.V.D. and Bauer, B. “*Extending UML for Agents*”, In Proceedings of the Agent-Oriented Information Systems Workshop (AOIS) at the 17th National Conference on Artificial Intelligence, Pages 3-17, Austin, Texas, 2000.

O'Malley, S. A. and DeLoach, S. A., “*Determining when to use an agent-oriented software engineering*”, In Proceedings of the Second International Workshop On Agent-Oriented Software Engineering (AOSE-2001), Pages 188-205, Montreal, 2001.

Omicini, A., “*SODA: Societies and Infrastructures in the Analysis and Design of Agent-Based Systems*”, in agent-Oriented Software Engineering, LNAI 1957, Springer-Verlag, Pages 185-194, Berlin, 2001.

Omicini, A. and Denti, E., “*From tuple spaces to tuple centres*”, Science of Computer Programming, 41(3): Pages 277–294, 2001.

Omicini, A. and Denti, E., “*Formal ReSpecT*”, Electronic Notes in Theoretical Computer Science, 48, Pages 179–196, 2001.

Omicini, A. and Zambonelli, F., “*Coordination for Internet application development*”, Int. J. of Autonomous Agents and Multi-Agent Systems, 2(3). Pages 251–269, 1999.

Padgham, L. and Winikoff, M., “*Prometheus: A Methodology for Developing Intelligent Agents*”, In Agent-Oriented Software Engineering III, LNCS 2585, Springer-Verlag, Pages 174-185, Berlin, 2003.

Padgham, L. and Winikoff, M., “*Prometheus: A pragmatic methodology for engineering intelligent agents*”, In Proceedings of the OOPSLA 2002 Workshop on Agent-Oriented Methodologies, Pages 97-108, Seattle, 2002.

Parsons, S. and Giorgini, P., “*On using degrees of belief in BDI agents*”, International Conference on Information Processing and Management of Uncertainty in Knowledge Based Systems, Paris, 1998.

Prasse, M., “*Evaluation of object-oriented modelling languages: A comparison between OML and UML*”, In Martin Schader and Axel Korthaus, editors, The Unified Modeling Language – Technical Aspects and Applications, Physica-Verlag, Pages 58-75, Heidelberg, 1998.

Pollack, M., “*The Use of Plans*”, Artificial Intelligence, Pages 43-68, 1992.

Rao, A.A. and Georgeff M.P., “*Modeling rational agents within a BDI-architecture*”, In Allen, J., Fikes, R. and Sandewall, E., editors. Proceedings of the Second International Conference on Principles of Knowledge Representation and Reasoning. Morgan Kaufmann Publishers, San Mateo, Ca, 1991.

Rao, A.S., “*AgentSpeak (L): BDI agents speak out in a logical computable language*”, In Proceedings of the Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW-96), Lecture Notes in Artificial Intelligence Vol. 1038, Springer-Verlag, Pages 42-55, 1996.

- Rao, A. S. and Georgeff, M. P., “*BDI Agents: From Theory to Practice*”, In Proc. 1st Int. Conf. on Multi-Agent Systems (ICMAS'95), San Francisco, CA, Pages 312–319, 1995.
- Riecken, D., “*An architecture of integrated agents*”, Communications of the ACM, 37(7), Pages 107–116, 1994.
- Rumbaugh, J., “*Notation notes: Principles for choosing notation*”, Journal of Object-Oriented Programming (JOOP), 8(10) Pages 11-14, 1996.
- Russell, S. and Norvig, P., “*Artificial Intelligence: A Modern Approach*”, Prentice-Hall, 1995.
- Rys, A., Jedrzejek, C. and Figaj, A., “*Use Case Maps as an Extension of UML for System Integration and Verification*”, At the 19th European Conference on Object Oriented Programming (ECOOP 2005) Glasgow, UK ACE, 2005.
- Sabas, A., Delisle, S. and Badri, M., “*A Comparative Analysis of Multiagent System Development Methodologies, Towards a Unified Approach*”, AT2AI-3, Vienna, Austria (EU), 2002.
- Sales, I. and Probert, R., “*From High-Level Behaviour to High-Level Design: Use Case Maps to Specification and Description Language*”, Submitted to SBRC'2000, 18th Brazilian Symposium on Computer Networks, Belo Horizonte, Brazil, 2000.
- Shehory, O. and Sturm, A., “*Evaluation of modeling techniques for agent-based systems*”, In Jorg P. Miiller, Elisabeth Andre, Sandip Sen, and Claude Frasson, editors, Proceedings of the Fifth International Conference on Autonomous Agents, Pages 624-631, ACM Press, 2001.
- Shoham, Y., “*Agent-oriented programming*”, Artificial Intelligence, 60(1), Pages 51-92, 1993.
- Schreiber, G., Akkermans, H., Anjewierden, A., Hoog, R. d., Shadbolt, N. R. and Wielinga, B., “*Knowledge Engineering and Management: The CommonKADS Methodology*”, the MIT Press, Massachusetts, 2000.
- Shoham, Y., “*An Overview of Agent-Oriented Programming*”, Software Agents, Bradshaw, AAAI Press, Menlo Park, CA, USA, 1997.
- Silva, C., Tedesco, P., Castro, J. and Pinto, R., “*Comparing Agent Orient-d Methodologies Using the NFR Approach*”, 3rd International Workshop on Software Engineering for Large-Scale Multi-Agent Systems (SELMAS), 2004.
- Sirbu, M. and Tygar, J.D., “*NetBill: An Internet Commerce System Optimized for Network-Delivered Services*”, IEEE Personal Communications COMPCON, Pages 20-25, 1995.
- Sturm, A. and Shehory, O., “*A Framework for Evaluating Agent-Oriented Methodologies*”, Workshop on Agent-Oriented Information System (AOIS), Melbourne, Australia, AOIS, 2003.
- Sturm, A., Dori, D. and Shehory, O., “*Single-Model Method for Specifying Multi-Agent Systems*”, The Second International Joint Conference on Autonomous Agents and Multi-Agent Systems, Melbourne, Australia, 2003.

Subrahmanian, V., Bonatti, P., Dix, J., Eiter, T., Kraus, S., Özcan, F. and Ross, R., “*Heterogenous Active Agents*”, MIT-Press, 2000.

Sudeikat, J., Braubach, L., Pokahr, A. and Lamersdorf, W., “*Evaluation of Agent - Oriented Software Methodologies - Examination of the Gap Between Modeling and Platform*”, Agent-Oriented Software Engineering V, Fifth Int. Workshop AOSE, Springer Verlag, 2004.

Taylor, Kujawski A. and Kowalski P., “*Naive Psychological Science: The Prevalence, Strength, and Sources of Misconceptions*”, The Psychological Record Vol. 54, Pages 15-25, 2004.

Thielscher, M., “*FLUX: A logic programming method for reasoning agents*”, Theory and Practice of Logic Programming, 2005.

Tsohatzidis, S. L., “*Foundations of Speech Act Theory: Philosophical and Linguistic Perspectives*”, London, Routledge, 1994.

Tran, Q.-N.N. and Low, G.C., “Comparison of ten agent-oriented methodologies”, In Agent-Oriented Methodologies, eds. B. Henderson-Sellers and P. Giorgini, Pages 341-367, Idea Group Publishing, Hershey, USA, 2005.

UML Specification v. 1.1 (OMG document ad/97-08-11), 1997.

UML Use case diagrams. : In http://atlas.kennesaw.edu/~dbraun/csis4650/A&D/UML_tutorial/use_case.htm.

Wood, B., Pethia, R., Gold, L. R. and Firth, R., “*A guide to the assessment of software development methods*”, Technical Report 88-TR-8, Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, PA, 1988.

Wood, M. F. and DeLoach, S. A., “*An Overview of the Multi-agent Systems Engineering Methodology*”, In Agent-Oriented Software Engineering, LNAI 1957, Springer-Verlag, Pages 207-222, Berlin, 2001.

Wooldridge, M., “*The Logical Modelling of Computational Multi-Agent Systems*”, PhD thesis, Department of Computation, UMIST, Manchester, UK, 1992.

Wooldridge, M. J., “*Introduction to Multi-agent Systems*”, John Wiley and Sons, 2002.

Wooldridge, M. and Jennings, N., “*Intelligent agents: Theory and practice*”, The Knowledge Engineering Review, 10(2), 1995.

Wooldridge, M. J., Jennings, N. R. and Kinny, D., “*The Gaia methodology for agent-oriented analysis and design*”, Autonomous Agents and Multi-Agent Systems, 3(3), Pages 285-312, 2000.

Yu, E. “*Agent-Oriented Modelling: Software Versus World*”, In Proceedings of the AOSE-2001, LNAI, 2222. Springer-Verlag, Pages 206-225, 2002.

Yu, E., “*Modelling Strategic Relationships for Process Reengineering*”, Ph.D. thesis, Department of Computer Science, University of Toronto, Canada, 1995.

Yu, E. and Mylopoulos, J., “*Understanding 'Why' in Software Process Modeling, Analysis, and Design*”, InProc. of the 16th IEEE International Conference on Software Engineering, Pages 159-168, Sorrento, Italy, 1994.

Zambonelli, F., Jennings, N. R. and Wooldridge, M., “*Developing multi-agent systems: The Gaia methodology*”, ACM Transactions on Software Engineering and Methodology, 12(3), Pages 317-370, 2003.

ZEUS, 1999 <http://www.labs.bt.com/projects/agents/zeus/index.htm>.