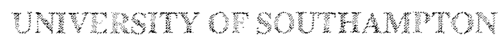

UNIVERSITY OF SOUTHAMPTON

Increasing Accessibility in Agent-Oriented Methodologies

by

Jorge Gonzalez-Palacios

A thesis submitted in partial fulfillment for the
degree of Doctor of Philosophy

in the
Faculty of Engineering, Science and Mathematics
School of Electronics and Computer Science

September 2006

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF ENGINEERING, SCIENCE AND MATHEMATICS
SCHOOL OF ELECTRONICS AND COMPUTER SCIENCE

Doctor of Philosophy

Increasing Accessibility in Agent-Oriented Methodologies

by Jorge Gonzalez-Palacios

The computing world is rapidly changing from a one in which a centralised approach is taken to one in which a highly distributed approach is taken, thus requiring software systems that operate in open, dynamic and heterogeneous environments. This has significantly increased the complexity of software systems, and has required the development of new paradigms for software development, such as the multi-agent approach to system development.

However, even though there is evidence of the suitability of the multi-agent approach to cope with the complexity of current systems, its use is not widespread in other areas of computing science, nor in industrial and commercial environments. This can be explained, particularly for agent-oriented methodologies, by the absence of key software engineering best practices. In particular, we have identified three groups of drawbacks that limit the use of agent-oriented methodologies: incomplete coverage of the development cycle, a lack of tools for supporting the development process, and a high degree of dependence on specific toolkits, methods or platforms. Although these issues negatively affect the applicability of the multi-agent approach in general, it is arguably for *open* systems that their effect is particularly noticeable.

In this thesis, therefore, we aim to address the issues involved in taking existing agent-oriented methodologies to a point where they can be effectively applied to the development of open systems. In order to do so, we consider the combination of organisational design and agent design, as well as the methodological process itself.

Specifically, we address organisational design by constructing a software engineering technique (software patterns) for the representation and incorporation of standard organisations into the organisational design of a multi-agent system. The agent design aspect is addressed by constructing an agent design phase which uses standard agent architectures through a pattern catalogue. Based on this, we develop a methodological process that combines the organisational and agent designs, and that also considers the use of iterations for making the development of a system more agile. This methodological process is exemplified and assessed by means of a case study. Finally, we address the problem of monitoring the correct behaviour of agents in an open system, by constructing a model for the specification of open multi-agent systems.

Contents

Acknowledgements **12**

1 Introduction **14**
1.1 Modern computing . . . 14
1.2 Agent-based computing . . . 15
1.3 Agent-oriented software engineering . . . 15
1.4 Drawbacks . . . 16
1.5 Aims . . . 17
1.6 Overview . . . 18

2 Agent-oriented software engineering **20**
2.1 Introduction . . . 20
2.2 Agents and multi-agent systems . . . 21
2.3 Agent architectures . . . 22
2.3.1 The subsumption architecture . . . 24
2.3.2 PRS . . . 24
2.4 Organisations in multi-agent systems . . . 26
2.5 Agent oriented software engineering . . . 28
2.5.1 Requirements engineering . . . 29
2.5.2 Languages . . . 29
2.5.3 Modelling languages . . . 30
2.5.3.1 AUML . . . 31
2.5.3.2 Others . . . 33
2.5.4 Platforms . . . 33
2.5.4.1 JADE . . . 34
2.5.4.2 ZEUS . . . 34
2.5.4.3 Industry-oriented platforms . . . 35
2.5.5 Methodologies . . . 36
2.6 Representative agent-oriented methodologies . . . 38
2.6.1 Gaia . . . 38
2.6.2 KGR . . . 39
2.6.3 MAS-CommonKADS . . . 40
2.6.4 MaSE . . . 41
2.6.5 Tropos . . . 42
2.6.6 INGENIAS . . . 43
2.6.7 Others . . . 44
2.7 Evaluation of methodologies . . . 45

2.7.1 Development process 45
2.7.2 Facilitators 46
2.8 Drawbacks in current methodologies 48
2.8.1 Supported phases 50
2.8.2 Agent architectures 50
2.8.3 Interactions in open systems 50
2.8.4 Iterative development 51
2.9 Conclusions 51

3 Modelling organisational structures **54**
3.1 Introduction 54
3.1.1 Interactions and organisations 54
3.1.2 Organisations in multi-agent systems 55
3.1.3 Organisation type selection 56
3.1.4 Organisational patterns 57
3.1.5 Overview 58
3.2 Gaia as the basis for the methodological approach 58
3.2.1 The main models of Gaia 59
3.2.1.1 Role model 59
3.2.1.2 Interaction model 60
3.2.1.3 The model of organisational rules 62
3.2.2 The Gaia process 62
3.2.2.1 The analysis phase 62
3.2.2.2 The architectural design phase 63
3.2.2.3 The detailed design phase 64
3.2.3 Discussion 64
3.3 Organisational rules 65
3.3.1 Organisations 65
3.3.2 Overview 66
3.3.3 Requirements of a language for organisational rules 70
3.3.4 A language for organisational rules 71
3.4 Organisational structures 75
3.4.1 Introduction 75
3.4.2 Characterisation and informal analysis 76
3.4.3 A model for organisational structures 76
3.4.3.1 Characterisation of organisational structures 77
3.4.3.2 Characterisation of control relationships 79
3.4.3.3 Language for expressing organisational structures 80
3.4.3.4 Example 81
3.4.3.5 Conclusions 82
3.5 Organisational Patterns 82
3.5.1 Introduction 82
3.5.2 Pattern layout 83
3.5.2.1 Pattern requirements 83
3.5.2.2 An organisational pattern layout 84
3.6 Catalogue of patterns 86
3.6.1 Introduction 86

3.6.2 The pipeline pattern . . . 87
3.6.3 The Simple hierarchy pattern . . . 93
3.6.4 The Marketplace pattern . . . 97
3.6.5 Selecting patterns . . . 105
3.6.6 Summary . . . 106
3.7 Related work and conclusions . . . 107

4 Modelling the internal structure of an agent **110**
4.1 Introduction . . . 110
4.2 Internal representation of agents . . . 112
4.2.1 Obtaining a detailed design from a high-level design . . . 112
4.2.2 Pattern layout . . . 113
4.3 The Subsumption architectural pattern . . . 114
4.3.1 The subsumption architecture . . . 115
4.3.2 Pattern description . . . 115
4.3.3 Obtaining a detailed design for the subsumption architecture . . . 119
4.4 The dMARS architectural pattern . . . 122
4.4.1 The dMARS architecture . . . 122
4.4.2 Pattern description . . . 123
4.4.3 Obtaining a detailed design for the dMARS architecture . . . 127
4.5 The TouringMachines architectural pattern . . . 129
4.5.1 The TouringMachines architecture . . . 130
4.5.2 Pattern description . . . 131
4.5.3 Obtaining a detailed design for the TouringMachines architecture . . . 134
4.6 Towards a general pattern . . . 137
4.6.1 Guiding the development of an architectural pattern . . . 138
4.7 Related work . . . 142
4.8 Conclusions . . . 143

5 An incremental and iterative methodological process **145**
5.1 The approach . . . 147
5.2 Requirements analysis . . . 149
5.3 Analysis . . . 150
5.3.1 Artefacts . . . 150
5.3.2 Activities . . . 151
5.4 Organisational design . . . 151
5.4.1 Artefacts . . . 152
5.4.2 Activities . . . 153
5.4.2.1 Defining the organisational structure . . . 153
5.4.2.2 Creating the organisational design models . . . 155
5.5 Agent design . . . 155
5.5.1 Artefacts . . . 156
5.5.1.1 The structure model . . . 156
5.5.1.2 The functionality model . . . 157
5.5.2 Activities . . . 157
5.5.2.1 Determining the agent architecture . . . 159
5.5.2.2 Creating class diagrams . . . 159

5.5.2.3 Creating scenarios . . . 160
5.6 Implementation . . . 160
5.7 Iterations . . . 161
5.8 Related work and conclusions . . . 162

6 Case study **164**
6.1 Problem statement . . . 164
6.2 Iterations . . . 166
6.3 First iteration . . . 167
6.3.1 Analysis . . . 167
6.3.1.1 Organisation model . . . 167
6.3.1.2 Environmental model . . . 167
6.3.1.3 Preliminary role and interaction models . . . 168
6.3.1.4 Preliminary rule model . . . 172
6.3.2 Organisational design . . . 174
6.3.2.1 Organisational structure . . . 174
6.3.2.2 Role model . . . 175
6.3.2.3 Interaction model . . . 178
6.3.2.4 Rule model . . . 179
6.3.3 Agent design . . . 180
6.3.3.1 Structure model . . . 181
6.3.3.2 Functionality model . . . 182
6.4 Second iteration . . . 183
6.4.1 Analysis . . . 183
6.4.2 Organisational design . . . 185
6.4.3 Agent design . . . 188
6.5 Conclusions . . . 188

7 Specification and Integrity in the Development of Open Systems **189**
7.1 Introduction . . . 189
7.2 Basic concepts . . . 191
7.2.1 A layered model for open multi-agent systems . . . 191
7.2.2 Roles . . . 194
7.2.3 Protocols . . . 194
7.2.4 Organisational rules . . . 195
7.3 Specification of open multi-agent systems . . . 197
7.3.1 General concepts model . . . 198
7.3.2 Participants model . . . 198
7.3.3 Interactions model . . . 199
7.3.4 Social constraints model . . . 202
7.3.5 Summary . . . 203
7.3.6 A model of open systems . . . 204
7.3.7 Ensuring information consistency . . . 206
7.4 Compliance monitoring . . . 208
7.4.1 Static analysis on agent entry . . . 208
7.4.2 Run-time participants analysis . . . 209
7.4.3 Run-time protocol analysis . . . 210

7.4.3.1 Algorithm for matching the head 210
7.4.3.2 Algorithm for matching the messages 212
7.4.4 Run-time organisational rules analysis 215
7.4.5 A design for checking static conditions 216
7.5 Conclusions and further work 219

8 Conclusions and future work **225**
8.1 Summary 225
8.2 Contributions 226
8.3 Limitations 228
8.4 Future work 229
8.5 Concluding remarks 230

A The Conference Management System problem statement **242**

List of Figures

2.1 The subsumption architecture 25
2.2 Main components of the PRS architecture 26
2.3 Example of an AUML sequence diagram 32
2.4 The AUML connectors 32
2.5 Example of nested protocol in AUML (after [66]) 32

3.1 Organisation of the Conference Management System 56
3.2 The models of the Gaia methodology (modified from [134]) 60
3.3 Example of a role schema 61
3.4 A generic protocol definition 61
3.5 Example of a protocol definition 62
3.6 Components of an organisation 66
3.7 Topology representation 76
3.8 Nodes linked by more than one arc 78
3.9 Using arrows to denote control relationships 78
3.10 Graphical representation of a pipeline structure 81
3.11 The Filter role 89
3.12 The Pipe role 89
3.13 Topology of a pipeline structure 90
3.14 The environmental entities of the pipeline structure 90
3.15 The GetInput protocol 91
3.16 The SupplyOutput protocol 91
3.17 Dynamics of the pipeline structure 92
3.18 The simple hierarchy structure 93
3.19 The Head role 95
3.20 The Leaf role 96
3.21 Main dynamics of the simple hierarchy structure 97
3.22 Roles in a marketplace, part 1 100
3.23 Roles in a marketplace, part 2 100
3.24 Roles in a marketplace, part 3 101
3.25 Roles in a marketplace, part 4 101
3.26 Roles in a marketplace, part 5 102
3.27 Roles in a marketplace, part 6 102
3.28 Topology of a market 103
3.29 The Sale scenario of the marketplace structure 104
3.30 The Entrance scenario of the marketplace structure 104
3.31 The role of organisational patterns in the methodology 106

4.1 Class diagram for subsumption architecture 117
4.2 Dynamics for subsumption architecture 118
4.3 Packages for the dMARS architecture 125
4.4 Class diagram for the dMARS architecture 126
4.5 Sequence diagram for the case when the event buffer is empty 128
4.6 Sequence diagram for the case when the event buffer is non-empty 129
4.7 The TouringMachines architecture 130
4.8 Packages in the TouringMachines architecture 132
4.9 Class Diagram of the Planning Layer 133
4.10 Main flow of operation in the TouringMachines architecture 133
4.11 Modelling the structure of effectors 140
4.12 Modelling the dynamics of effectors 140
4.13 Modelling sensors 140

5.1 The workflows in the process and their artefacts 147
5.2 Workflows and iterations of the process 148
5.3 Workflows of the process 149
5.4 Activities of the analysis 152
5.5 Activities of the organisational design 154
5.6 Example of a class diagram in the structure model 157
5.7 The structure model 158
5.8 The artefacts of the agent design 158
5.9 The activities of the agent design 159

6.1 Communication paths in the case study 169
6.2 Preliminary roles, part 1 169
6.3 Preliminary roles, part 2 170
6.4 Preliminary roles, part 3 170
6.5 Preliminary roles, part 4 171
6.6 Preliminary roles, part 5 171
6.7 Preliminary interaction protocols, part 1 172
6.8 Preliminary interaction protocols, part 2 172
6.9 Preliminary interaction protocols, part 3 172
6.10 Preliminary interaction protocols, part 4 173
6.11 Preliminary interaction protocols, part 5 173
6.12 Preliminary interaction protocols, part 6 173
6.13 Preliminary interaction protocols, part 7 173
6.14 Preliminary organisational structure 175
6.15 Organisational structure 175
6.16 Role model, part 1 176
6.17 Role model, part 2 176
6.18 Role model, part 3 177
6.19 Role model, part 4 177
6.20 Interaction model, part 1 178
6.21 Interaction model, part 2 178
6.22 Interaction model, part 3 178
6.23 Interaction model, part 4 178

6.24 Interaction model, part 5 179
6.25 Interaction model, part 6 179
6.26 Class diagram of the structure model 182
6.27 Sequence diagram of the functionality model 183
6.28 Preliminary protocol description of registering users 184
6.29 Preliminary protocol description of expelling users 184
6.30 Preliminary role description of the community clerk 184
6.31 Preliminary role description of the personal clerk 185
6.32 Role description of the personal clerk 186
6.33 Role description of the community clerk 186
6.34 Protocol description of registering users 187
6.35 Protocol description of expelling users 187

7.1 Layered decomposition of open multi-agent systems 192
7.2 Example of a protocol specification 196
7.3 Organisational rules for the Conference Management System 196
7.4 The general form of the General Concepts Model 199
7.5 The application of the General Concepts Model to the CMS example 200
7.6 The paper concept 201
7.7 The general form of the participants model 201
7.8 The application of the Participants Model to the CMS example 202
7.9 The general form of the interactions model 202
7.10 The application of the Interactions Model to the CMS example 203
7.11 The general form of the social constraints model 203
7.12 The application of the Social Constraints Model to the CMS example 203
7.13 Description of the UpdateCall service 204
7.14 The function of the monitor 209
7.15 Different Connectors in AUML 211
7.16 **Algorithm:** MATCHING THE HEAD 212
7.17 **Algorithm:** MATCHING THE MESSAGES 214
7.18 Example of sequence of messages 214
7.19 **Algorithm:** CHECKING STATIC ORGANISATIONAL RULES 216
7.20 Data structure of agents and their roles 217
7.21 Information needed by the monitor 218
7.22 Monitor's database 218
7.23 Components of the monitor 219
7.24 Checking protocol compliance 220
7.25 **Algorithm:** TRANSFORMING A SEQUENCE DIAGRAM INTO A FSM 221
7.26 The operation of the warden 223

List of Tables

2.1 Covered phases 47
2.2 Facilitators 48

3.1 Characterisation of roles in Gaia 60
3.2 Characterisation of services in Gaia 64
3.3 Temporal operators 68
3.4 Practical operators 68
3.5 Grammar for the LEVOR language 72
3.6 Pre-defined functions in the LEVOR language 73
3.7 Pre-defined predicates in the LEVOR language 73
3.8 Summary of notation 78
3.9 Pre-defined control relationships 78
3.10 Formal definition of a pipeline 81
3.11 Summary of the layout for describing organisational patterns 85
3.12 Environmental entities of a pipeline and their rights of access 91
3.13 Environmental entities of a Marketplace and their rights of access 103

6.1 Iteration decomposition of the case study 167

7.1 Summary of notation 207
7.2 Inputs for the protocol checking procedures 211

Acknowledgements

In the first place, I would like to thank my supervisor, Michael Luck, for his advice, patience and motivation. His support has been crucial during the elaboration of this thesis. I also thank Terry Payne, Chris Reed and Nick Jennings for their valuable comments and suggestions. I am grateful to the Mexican Council for Science and Technology (Conacyt) for providing financial assistance for most of the duration of my PhD studies. I thank my parents and relatives for their unconditional financial and emotional support. Last, but not least, I wish to thank those friends who made my stay in Southampton a more enjoyable experience: Fabiola, Cora, Alex, Arturo, Suzana, and many others. In particular, I thank Rocio, for her help with printing and binding.

A Viridiana y Carmen, por ser fuente de motivación y acompañarme en esta aventura.

A Josefina y Joel, esto es suyo, al igual que mi corazón.

Chapter 1

Introduction

1.1 Modern computing

The last few years have witnessed a dramatic change in the computing world, which has shifted from a centralised approach to a highly distributed one. This has been caused, arguably, by the explosive increase in the number of computers and the networks they form. As a consequence, the complexity of computing systems has increased considerably. In particular, highly distributed environments have increased the complexity of software systems because of the heterogeneity, openness, and dynamism associated with them.

This increase in the complexity of software systems has also had a considerable impact on the way software is developed, resulting in the problem of how to build systems which are much more complex, but keeping to the same time and budget constraints and, at the same time, maintaining high quality levels. Several approaches have been used to solve this problem, among them the use of more powerful, although traditional[1], software methodologies and tools. Although such approaches have achieved some success in modelling the features found in highly distributed systems — dynamic, heterogeneous and open environments — traditional approaches have proven to be complicated, error prone and time-consuming. The reason for this is that the concepts on which traditional software engineering is based are not at the level of abstraction required to model complex systems [75]. For instance, *pro-activeness* and *autonomy* (explained in Section 1.2) have been recognised as valuable modelling tools, but objects, by themselves, are not capable of exhibiting pro-activeness, since objects are passive entities that operate at the request of other objects. Similarly, objects are incapable of *autonomy*, since objects cannot choose which other objects can access their public services.

The difficulty in overcoming these limitations by means of traditional approaches has led to the search for new paradigms that cope successfully with the complexity of current software. Some

[1]In this context, we use *traditional* to refer to software paradigms commonly used in the development of software systems, namely structured methods and object-based approaches.

of the new paradigms that have been used with certain success are extreme programming [8] and, more generally, agile software development [19]. The idea behind these paradigms is to develop the *correct* system in the *right* time by delivering and testing versions of the system from the very early stages, for obtaining user feedback, and incorporating the suggested changes. However, although these paradigms offer improvements in the methodological process, in general they maintain the same basic modelling abstraction, namely objects, and so they suffer from the same limitations as other object approaches.

1.2 Agent-based computing

In contrast, the multi-agent approach is a new paradigm based on a different modelling abstraction, known as an *agent*. In this paradigm, a software system is considered as composed of several agents that interact according to a high-level discourse in order to achieve individual or overall goals. There are two aspects in this view that differentiate the multi-agent approach from other approaches, and need further explanation.

First, the *high-level discourse* means that the interaction between agents is in terms of the problem domain, rather than in terms of low-level communication protocols or procedure calls. This allows agents to focus on the problem solution — rather than on managing communication details — and to have a better model of the real world situation.

Second, agents are the distinguishing aspect of this approach for which, however, there is no consensus on their exact meaning. From an engineering perspective, nevertheless, it is convenient to consider an agent as a software system that exhibits autonomy, pro-activeness and sociality. Autonomy is a twofold concept, denoting the capability of acting without human intervention, as well as the capability of deciding which actions to commit to. Pro-activeness refers to the capability of pursuing goals. Last, but not least, the sociality property refers to the ability of an agent to interact with other agents in order to achieve its goals.

The multi-agent approach has been successfully applied in the development of complex distributed systems, in areas such as manufacturing systems [37], air-traffic control [104], and electronic markets [17]. Additionally, practical evidence suggests that multi-agent systems are suitable for developing the type of applications that emergent technologies — such as the Grid [42] and ubiquitous computing [1] — require.

1.3 Agent-oriented software engineering

The use of the multi-agent approach to engineering software systems, known as agent-oriented software engineering (AOSE), is a relatively new discipline that has had, nonetheless, a rapid evolution. Nowadays, AOSE considers practically all the components of software development,

including programming languages [103, 23], methodologies [134, 25, 9, 55], platforms of operation [70, 120], and formal methods [90]. Of all these components, methodologies are perhaps one of the most important because they determine how the other components are used during the development of a system. More generally, software methodologies provide a way to engineer systems in an efficient, repeatable, robust and controllable fashion, thus helping to reduce development costs and to increase software quality.

It is, arguably, because of this importance that a large number of agent-oriented methodologies have been proposed to date [67, 92, 5]. Indeed, existing methodologies cover a broad range of applications, and vary in several aspects, such as the concepts they use to model multi-agent systems, the development activities they cover, the types of applications they are targeted at, and their degree of openness for using different tools and technologies.

1.4 Drawbacks

However, in spite of their number and variety, agent-oriented methodologies are not completely suitable for use in commercial and industrial environments — as well as in mainstream computing — mainly because of their immaturity from a software engineering viewpoint. In other words, the absence of recognised software practices and principles in agent-oriented methodologies has originated serious drawbacks, thus preventing their use in larger communities of developers.

Most of the drawbacks found in agent-oriented methodologies can be classified as one or more of the following groups (a more detailed review is presented in Chapter 2).

1. Incomplete coverage of the development cycle. Several methodologies consider only some of the phases of the whole development cycle, typically analysis and design of agent interactions. This seems reasonable if we consider that these phases form the core of the multi-agent approach (the other core phase, the design of agents, has partially been explored in the study of agent architectures). However, although some methodologies have *intentionally* left out certain phases, a complete process is needed in order to build real world systems.

2. Lack of tools for supporting development activities. Here, we refer to several types of tools, including: tools for graphical design; integrated development environments (IDEs) for analysis, design and code generation; libraries of common design solutions; and code debuggers. Without these tools, many of the development tasks become unnecessarily time-consuming and error prone, thus discouraging the use of the methodology in question, and even the multi-agent approach itself.

3. Dependence on specific toolkits, methods or platforms. Several methodological processes rely strongly on specific toolkits (for example ZEUS [120] and Jade [70]), agent architectures, or platforms (for example FIPA [35] implementations). This dependence has the

advantage of facilitating the learning process for those practitioners already familiar with those components, and of alleviating the problem of carrying out a selection. However, the effectiveness of a methodology is reduced by the impossibility of selecting the best solution for each application.

1.5 Aims

Current methodologies for multi-agent systems are largely focused around the development of closed systems, in which all their components are known in advance. This, however, is not usually observed in highly distributed systems. The aim of this thesis, therefore, is to address the issues involved in taking existing methodologies to a point where they can be effectively applied to the development of *open* systems. Here, an open system is one in which agents are not designed in common, do not share a common goal, are possibly developed by different development teams, and whose composition varies by the incorporation of new agents into the system or by their exit from the system [134]. To do so, we need to consider the combination of macro-level (organisational design) and micro-level (agent design) aspects of methodologies, as well as the methodological process itself.

The specific aims of this thesis are as follows.

- Incorporate agent design into methodologies. The agent design phase is an essential part of a complete methodological process that is, nevertheless, absent from many current methodologies and, when present, is usually tied to particular agent architectures. This is inadequate for open systems, since it restricts the selection of the best architecture for a particular agent, and violates the *open* nature of these systems. In this thesis, we aim to provide a methodological process for designing the agents of a system.

- Represent organisations. Organisations are an appropriate means for modelling the structure of multi-agent systems. Organisations, however, are usually described in agent-oriented methodologies by using informal methods, such as plain English or figures, which produces inexact descriptions. In this thesis, we aim to develop a model for exact and complete representation of the components of an organisation, including topology, control regime and organisational rules.

- Construct a software engineering technique for incorporating standard organisations into the architectural design of a multi-agent system. While the architecture of a multi-agent system can be modelled by means of an organisation, it is not always easy to determine the organisation that best models a particular system. In this thesis, we aim to develop some means for determining the type of organisation that best suits the characteristics of a given multi-agent system.

- Construct a software engineering technique for supporting the design of agents by the use of standard agent architectures. The design of agents relies on the use of models for specifying the internal structure and operation of an agent, and these are precisely the types of models that agent architectures provide. However, in spite of this natural correspondence, using agent architectures for engineering practical agents presents some obstacles, such as the mismatch between agent-based and software engineering abstractions. In this thesis, we will provide a technique that removes these obstacles and allows the incorporation of well known architectures to a methodological process.
- Create a model for the specification of open multi-agent systems. With the purpose of incorporating new agents into an open multi-agent system, it is important to specify the facilities provided by the system, as well as the restrictions for its use. However, this must be done in such a way that no assumptions are made about how the agents are actually implemented, since this is undetermined at design time. To facilitate this specification, we aim to describe a model that abstracts the properties of an open multi-agent system, and can be instantiated to obtain specifications for particular systems. A specification constructed in this way can also be used to check, at run-time, if the agents joining the system comply with its restrictions of use.

In seeking to achieve these aims, we adopt software engineering principles for making agent-oriented methodologies more agile. More specifically, in this thesis we aim to develop a methodological process that allows the construction of a system by means of incremental executable versions. Producing executable versions of a system from early stages of the development, is beneficial for obtaining rapid user feedback, and thus reducing the risk of producing the *wrong* system. It also helps to reduce development time, since it increases the parallelism among the development activities. The applicability of such a methodological process is assessed by means of a case study.

In summary, it is important to overcome the drawbacks found in current methodologies, in order to achieve widespread use of the multi-agent paradigm, and consolidate it as a real option for the development of open systems. A possible approach for overcoming these drawbacks is by considering that current agent-oriented software engineering contains much valuable work, and what is missing is the introduction, or reinforcement, of recognised software practices. Thus, in this thesis, we aim to provide a means for completing and incrementing the maturity of agent-oriented methodological processes, so that they can adequately cope with the construction of such systems.

1.6 Overview

The rest of the thesis is organised as follows. Chapter 2 justifies the agent-oriented approach as a valuable tool for systems development, and presents the state of the art of AOSE. In Chapter 3,

we present a framework for constructing organisational patterns, which are representations of standard organisations for facilitating the organisational design of multi-agent systems. Chapter 4 deals with modelling internal composition of agents. We address this problem by incorporating the use of agent architectures into the design processes. Although the results obtained in these chapters can be used on their own, they are more useful as part of a methodological process. Such a process, which also considers the use of iterations, is presented in Chapter 5. Chapter 6 describes a case study in which we apply several of the results obtained in the previous chapters. Chapter 7 addresses the problem of monitoring the correct operation of an open system at run-time. To this end, we present a specification template that, when instanced for a particular system, produces a specification whose compliance ensures the correct operation of the system. Finally, Chapter 8 presents our conclusions and indicates future work.

Chapter 2

Agent-oriented software engineering

2.1 Introduction

The agent-oriented paradigm views a software system as composed of autonomous, pro-active entities (agents) that interact to achieve overall goals. Although research in distributed artificial intelligence has shown that the paradigm is suitable for modelling complex systems running in dynamic environments [131], some form of *engineering* of agent-based applications is needed to encourage its use in other areas of computing, as well as in *real world* applications. This is because these require the development of applications in a systematic form which, at the same time, is sufficiently comprehensive for a practitioner with *average* skills. Traditionally, *software methodologies* have provided such a way of engineering computer applications. More specifically, methodologies are useful for developing applications in an efficient, repeatable, robust and controllable fashion, and in so doing they help to reduce development costs and to increase software quality.

Although initially some attempts were made to develop agent-based systems following traditional methods [80], nowadays it is agreed that a new approach is needed to take advantage of all the characteristics of agenthood. Current work on agent-oriented software engineering covers almost all the activities of software development, requirements engineering [130], analysis [126], design [24], implementation [94], and code generation [96]. However, most of this work can be considered experimental or at the exploration stage [124].

The aim of this chapter is twofold: to review the main concepts in the field of multi-agent systems, particularly those essential to understand the benefits, obstacles and solutions in applying the multi-agent approach for the development of software systems; and to present the state of the art of agent-oriented software engineering (AOSE). With this aim, we first introduce general concepts of agents and multi-agent systems in Section 2.2. Then, models for both aspects of multi-agent systems are reviewed, the internal composition of agents (Section 2.3) and the ways in which they interact (Section 2.4). Next, we move our attention to the state of the art in agent-oriented software engineering in Section 2.5. After that, the focus is put on methodologies,

by reviewing some representative cases in Section 2.6, presenting an evaluation in Section 2.7, and summarising their main drawbacks in Section 2.8. Finally, our conclusions are presented in Section 2.9.

However, agent-oriented software engineering is a large field that is impossible to cover here completely. For this reason, we focus our review on methodologies and directly related concepts. Despite this consideration, the literature available is still so vast that it is impossible to carry out any exhaustive review, so we only consider some representative cases.

2.2 Agents and multi-agent systems

Arguably, this chapter should begin with *the* definition of agenthood, since it deals with agent-oriented systems. However, since it is known that there is no *unique* definition of agenthood [129, 127, 89] (the interested reader is referred to [47] for a survey of definitions), any definition adopted at this point will constrain and bias what we can say about agent-based computing. Since the purpose of this chapter is to provide a general overview of the field, instead of presenting a definition of agenthood, we prefer to review different perspectives, leaving until Section 2.9 the definition we will adopt for the rest of the thesis.

Wooldridge and Jennings describe two viewpoints of agenthood [129]: a weak notion and a strong notion. The weak notion is the most popular among the mainstream computing community, especially software engineers, and views an agent as a UNIX-like process with properties such as autonomy, social ability, reactivity and pro-activity. Autonomy refers to the capabilities of agents to work without human intervention and to have control over their own states and actions. Social ability is the capability to communicate with other agents at a high level of discourse. Reactivity refers to the property of perceiving and responding, in a timely fashion, to changes in the environment. Finally, pro-activity deals with the capability of an agent to select its own goals and act according to them.

By contrast, the strong notion of agenthood is common among the artificial intelligence community, and views an agent as a computer system that, in addition to having the properties mentioned above, can be conceptualised or modelled as if it had human characteristics such as mentalistic notions like knowledge, belief, intention and obligation.

In addition to the weak and strong notions, we must also emphasise the existence of a different perspective which consists in viewing agents as an abstraction for modelling software systems [74, 127], since it is on this perspective that agent-oriented software engineering is based. Here, agents are used to model entities of the world, and software systems are viewed as sets of agents interacting to achieve the desired functionality. As a result of their characteristics, agents represent a promising means for developing applications populated by autonomous, pro-active and/or reactive entities. More generally, this suggests complex applications running in dynamic environments.

It should be noted that each of these perspectives is closely related to the others. For example, the relation between agents as a software abstraction and the weak notion of agenthood resembles that between *classes* and *instances of classes* in the object-based paradigm, in the sense that both refer to the same concept but differ in the phase they are used: run-time for the latter and design for the former. Furthermore, far from being mutually exclusive, the different perspectives complement each other, and it is this diversity in perspectives that has been recognised as a key factor for the popularity of agents [127, 89].

2.3 Agent architectures

The term *agent architecture* has been employed in the literature with different meanings. For our purposes, an agent architecture is a specific collection of software modules, typically designated by boxes with arrows indicating the data and control flow among modules [129].

Agent architectures can be classified into deliberative, reactive, and hybrid. The deliberative architectures were the first to appear and are characterised by an explicitly represented, symbolic model of the world; and by basing their decision processes on pattern-matching and symbolic manipulation techniques. Thus, some of the problems to tackle in these architectures are the translation of real world concepts into symbols, and an efficient and accurate representation of the decision process. In general, however, solutions to these problems require large amounts of computation, which makes deliberative architectures unsuitable for many practical problems. We will not elaborate this problems here, since they relate more to traditional artificial intelligence than to the agent approach, but the interested reader can refer to [52]. The difficulty of these problems has led to the creation of alternative (and specifically agent) approaches, such as reactive architectures and hybrid architectures.

Reactive architectures cover a broad range of approaches having in common the avoidance of any kind of central symbolic world model or complex symbolic reasoning. As a result of this, these types of agents may respond more quickly to changes in the environment. In spite of their simplicity, or perhaps because of it, it is not possible to use a reactive architecture to develop agents whose behaviours depend strongly on their execution history or on complex reasoning.

Hybrid architectures attempt to combine the best of deliberative and reactive architectures by having two (or more) separate components, at least one deliberative and one reactive. The reactive component deals with important events that need a quick response, while the deliberative component is in charge of planning and reasoning activities. However, the exact relationship of the components and the control between them depends on each specific architecture.

The belief desire intention (BDI) architectures [61, 128] are an important type of hybrid architecture. BDI architectures are representations of agents whose behaviour can be described as if they had mental attitudes of beliefs, desires and intentions. Beliefs represent the knowledge the

agents possess, desires describe the goals the agent pursues, and intentions are the committed plans chosen to pursue those goals.

BDI architectures form a *family* of architectures sharing the BDI stance but differing in the roles they assign to beliefs, desires and intentions in the functionality of the system, as well as the form in which they are represented and controlled. Although beliefs are well understood, the exact roles played by desires and intentions have been subject to controversy. While both desires and intentions refer to a state of affairs that an agent wants to bring about, in the case of intentions there must be a certain commitment to achieve them. Therefore, while an agent may have some desires, it might never set out to accomplish them. On the other hand, intentions cause an agent to act. There are several theories that attempt to describe, analyse and specify the behaviour of an agent by describing the relationships between beliefs, desires and intentions, and the motivation behind such theories is diverse. Thus, while some aim to explain and predict agent behaviour from an observer's perspective [33], some are used to design agent architectures [29], and others have been applied to social agents for supporting reasoning about other agents engaged in group activity [72].

Since they first appeared, BDI architectures have been popular among the agent community for three key reasons. First, it is more natural to us, as humans, to model systems based on intentional notions. Second, most BDI architectures have a well-founded philosophical and theoretical background. Finally, these architectures, arguably, are more flexible than purely deliberative or reactive ones. Being hybrid, BDI architectures lie between purely reactive and purely deliberative systems. For this reason, they cannot deliver the same performance as purely reactive systems when operating in highly dynamic environments, although their performance can be increased by tuning the reasoning and deliberation strategies to the specific requirements of the application in question.

BDI architectures have been successfully used not only in research, but also in commercial and industrial applications. For example, OASIS [104], whose development was based on a BDI architecture, — dMARS [29] — is an air-traffic management system that has been successfully tested at Sydney airport. Agents in OASIS are of two types: *aircraft agents* and *global agents*. Each aircraft agent is associated with an aircraft and is responsible for controlling its flight. By contrast, the global agents are responsible for the sequencing and coordination of aircraft agents. During run-time, up to 80 agents operate concurrently to give control directives to flow controllers on a real-time basis.

Arguably, the success of BDI architectures is due to the following aspects [104].

- The application programming is based on plan construction, facilitating modular and incremental development.
- The balance between reactive and goal-oriented behaviour is managed by the system, so that end users need not be involved in complex low level programming, which improves reliability.

- End users can encode their knowledge in terms of mental attitudes instead of low-level languages.

The number of different architectures proposed to date makes an exhaustive review impossible so, as discussed in the introduction, we restrict our review to some representative architectures. Also, in this review we do not consider deliberative architectures, since they are essentially concerned with *planning* from traditional artificial intelligence, and not with agent-based computing *per se*. Moreover, consideration of such architectures could easily double the length of the review and would not add to the content of this thesis. Below, therefore, we focus on other types of architectures.

2.3.1 The subsumption architecture

The subsumption architecture [10, 128] is a reactive architecture developed by Brooks, that bases its function on the existence of *behaviours* and their relationships of *inhibition*. Each behaviour is intended to achieve a specific task and associates perceptual inputs with actions. For example, in the case of a vehicle control application, the behaviour, *changing direction if an obstacle is found in front*, associates the perceptual input, *an obstacle is in front*, with the task, *change direction*. To pursue its aim, each behaviour continually senses the environment until the environmental state matches its associated perceptual input, in which case the associated action is performed. In this example, the environment is continually sensed until an obstacle is detected in front of the vehicle, in which case the action of changing direction is performed. However, since an environment state may match more than one behaviour, an *inhibition relation* is used to specify priorities. According to this inhibition relation, the behaviours are arranged into layers, with lower layers capable of inhibiting upper layers, and the higher the layer the more abstract its behaviour (as in Figure 2.1, which illustrates the relationship between perceptual input, behaviours and action, and in which the inhibition relation has been represented as dotted lines). For example, in the case of vehicle control, the behaviour corresponding to *collision avoidance* occupies a lower layer than that of the behaviour corresponding to *reach the destination*, since avoiding an obstacle has priority over reaching the destination.

2.3.2 PRS

The Procedural Reasoning System [51] (PRS) was originally developed as part of a NASA project, and is based on a well-founded theoretical background. Beliefs, desires and intentions in PRS are represented explicitly, and together determine the actions of the system. They are also dynamically modified by a reasoning mechanism. As indicated in Figure 2.2, PRS is composed of an interpreter and the following modules: *database*, *goal stack*, *knowledge area library*, and *intention structure*. The database contains current facts about the world, while the knowledge area library is a store of *knowledge areas* (KAs), which in turn are knowledge about how to accomplish tasks and how to react in certain circumstances.

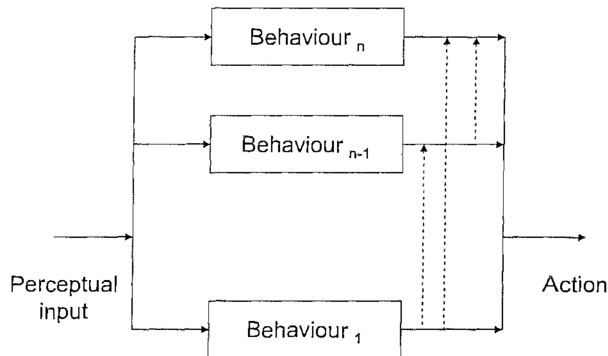

FIGURE 2.1: The subsumption architecture

A KA is composed of a body and an invocation condition. The body contains the steps of the procedure, and the invocation condition describes the circumstances under which the KA is applicable. Some of the KAs in the library are application specific, while others address the general management of PRS itself, such as choosing among relevant KAs.

In contrast to other BDI architectures, in PRS goals represent the desired behaviour of the system instead of static states of the world to be achieved. They are presented not only in the goal stack but also as part of the KAs. Intentions are tasks that the system has chosen to execute immediately or in the future, and consist of an initial KA and of other sub-tasks invoked in accomplishing the task. Such intentions are inserted into the intention structure, which is essentially a list with precedence.

For reasons of relevance and brevity, we will not elaborate in detail the operation of PRS, but briefly sketch it here only; we consider its successor dMARS in more detail in Chapter 4. In short, however, the interpreter controls the operation of the system, as follows. At a given time, the system has some goals and holds some beliefs, according to which some KAs become applicable, one of which is chosen to be executed and so is placed in the intention structure. While executing this KA, some goals are produced and placed in the goal stack. If new beliefs are acquired, consistency-checking procedures are applied. Also, new beliefs and new goals can activate some new KAs, in which case the interpreter can decide to perform some other goal. This has the effect of making the agent less committed to intentions and more aware of the environment. In fact, in PRS it is the KAs that provide a quick response to changes in the environment, forming the *reactive* component found in hybrid architectures. This is not a separate component of the architecture in this case.

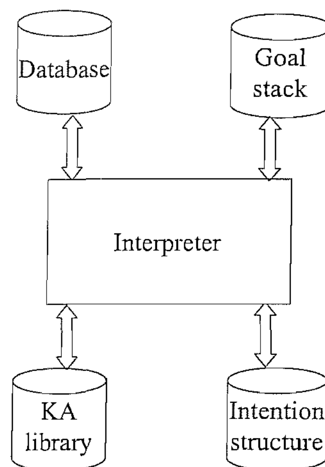

FIGURE 2.2: Main components of the PRS architecture

2.4 Organisations in multi-agent systems

While the previous section dealt with the internal aspects of agenthood, relating to how agents are internally composed, this section deals with the *interactions* between agents, or the ways in which they act together in order to solve common or inter-related problems.

Although several metaphors have been proposed to model the way agents interact in multi-agent systems [68, 71], the *organisational* metaphor [45] is emerging as one of the most utilised in agent-oriented software engineering [132, 82, 25], arguably because it is intuitive and has been successfully applied in several situations. Roughly, the organisational metaphor is based on how humans work together to solve problems in the context of an organisation, such as a business. A business has goals to achieve and in order to achieve them the goals are decomposed into specific tasks, like production and distribution. These specific tasks are assigned to roles that are played by humans. In order to carry out their tasks, roles interact according to pre-defined patterns, which define which roles are subordinated to the authority of others. These interactions also form a distinguishable network of communication paths.

More generally, according to the organisational metaphor, each agent in a multi-agent system can play one or more *roles*. Each role, in turn, is in charge of pursuing one or several well-defined *responsibilities*, which are fulfilled, generally, by *interacting* with other roles. However, an interaction between two roles is not only a relation of *association*, but in fact establishes a relation of *authority*, which is an integral part of the definition of the role. Roles, interactions and authority relationships define the *structure* of the organisation.

Organisations guide the way in which agents interact in a multi-agent system to achieve individual or global goals, and influence how they coordinate, allocate resources, and are subordinated

one to the other. Thus, by interacting with other agents, organisations help simple agents to achieve complex tasks, and sophisticated agents to reduce the complexity of their processes [64].

Arguably, all multi-agent systems have some form of organisation, even implicit, since in any multi-agent system distributed agents act together through relations, carry out assigned tasks, and use resources to accomplish them. However, more than one organisation might fit a particular system (e.g. a production line can be modelled as a pipeline, as well as a hierarchy [134]), and different organisations produce different system performance in terms of efficiency (e.g. communication and computation overhead), reliability and uncertainty management, since organisations differ in the way tasks are distributed and the communication paths they possess. Also, organisations present different levels of scalability, redundancy and flexibility.

Although no two organisations are identical, it is possible to group them in certain types, according to the *topology* their interactions form and the authority relationships their agents exhibit. The following are some of the most used types, or *paradigms*, of organisations.

Hierarchies [45, 64] This is one of the simplest and most studied form of organisation. In hierarchies, agents are conceptually arranged in a tree-like structure; the higher their position, the more important, in some sense, their role. Generally, lower-level agents produce data to feed higher-level agents, which in turn perform more complex processing such as consolidation, analysis or decision making. Hierarchies have been extensively used to model distributed applications [45, 135, 95]. In fact, the well known *contract net protocol* [115] tends to produce hierarchical structures. (Under a contract net protocol an agent can be assisted by other agents to complete its tasks, through assigning a subtask by advertising it, receiving offers and selecting the most convenient.)

Holarchies [38, 64] In this paradigm a system is viewed as composed of basic units of organisation, *holons*, which in turn can be seen as formed of other (more basic) holons; for example, a manufacturing system is composed of manufacturing units, which are in turn composed of devices, operators, processes, and so on. The key aspect of holarchies is the *partial* autonomy of holons, since the absence of autonomy would degenerate into a hierarchy and complete autonomy would lead to an unorganised group. More specifically, if the relationships between agents in a system are of complete subordination, the holarchy can degenerate into a hierarchy. Another important aspect of holarchies is that each of the holons represents a complex sub-organisation that can be decomposed further. Modelling a system into a hierarchical nested structure has proven to be suitable for modelling certain kinds of practical problems such as manufacturing control [37, 136]. For instance, in [136] a model combining different types of holons (static, mediator, and dynamic) is used to create an organisation for controlling manufacturing systems. Here, statically-created holons are used to represent entities of the environment such as manufacturing devices, design plans and conveyors, whereas dynamically-created holons are used to represent new tasks, and mediator-type holons manage orders and coordinate resources.

Teams A team [93, 7] is a system of cooperative agents that pursue a common goal. Since there is no restriction on the types of their interactions, team topologies tend to be quite arbitrary. However, it is often the case that members of a team share their *mental state*, particularly for common representations of shared goals, mutual beliefs and team-level behaviours. For example, Jennings [73] shows that, by sharing representations of common tasks, and through the progress of cooperation, agents are able to successfully solve electricity transportation problems, but they are prone to behaving incoherently in the lack of shared representations.

Markets [108, 125, 45] In this particular type of organisation, agents can buy and sell items such as goods, services or tasks. Agents playing the role of buyer place bids, and agents playing the role of seller receive the bids and determine the winner, in a manner that largely resembles a real-world marketplace. It is usually the case that some agents exist to facilitate the operation of the organisation in common tasks such as receiving bids and determining the winner. Kasbah [17] is one of the first examples of an agent-based marketplace, in which buyers and sellers describe the type of goods they are interested in by means of a list of features, a desired price and a threshold price. Here, a sale occurs when a there is a buyer willing to pay the price of the seller.

The way organisations are characterised differs for each particular approach. For example, in the Gaia methodology [132], an organisation of roles is composed of *structure* and *organisational rules*, where the structure of an organisation is described by its *topology* and *control regime*. The topology consists of the set of communication paths formed by the interaction of the roles in the organisation, and may take typical forms such as lines, trees or networks. On the other hand, the control regime encompasses the authority relations between the roles; for example, in an *employment* control regime, low-level roles are subordinated to high-level roles as the result of a work partition. Finally, organisational rules provide constraints on the way the elements of the organisation operate; for example, in a marketplace, an organisational rule could state that no product delivery can be made without receiving the corresponding payment.

In summary, the organisational approach applied to the multi-agent paradigm is a promising tool to cope with the complexity of current software systems [134, 25], due to the fact that organisations provide a conceptual framework in which the complex interactions carried out by agents can be appropriately modelled. This conceptual framework constitutes a layer of abstraction that is situated on top of, and complemented by, that provided by agents [88].

2.5 Agent oriented software engineering

Agent oriented software engineering (AOSE), which is concerned with engineering software systems having *agents* as the main design concept, is an evolving discipline whose aim is to provide methods, techniques and tools to facilitate the development of agent-based applications

in a repeatable, systematic and controlled way. We start this section by justifying the existence of AOSE and characterising the type of problems it attempts to cope with, and then proceed to present the current state of the art.

For the purpose of presentation, we divide AOSE into the following topics: requirements engineering; languages for programming, communication and coordination, and ontology specification; development tools and platforms; and methodologies for analysis, design, and implementation. Each of these is described below.

2.5.1 Requirements engineering

Requirements engineering deals with eliciting, modelling and analysing the functional and non-functional capabilities that a system should have. It is the front-end activity in the development process, and also plays an important role in the management of change in all phases. There are two non-mutually exclusive approaches in requirements engineering that are relevant to agent orientation: agent-oriented requirements engineering and goal-oriented requirements engineering, both considered below. Agent-oriented requirements engineering encompasses several approaches that primarily rely on the concept of agents, examples of which are i* [130] and ALBERT [30]. i* is a modelling framework based on the concept of agents with intentional properties such as goals and commitments, while ALBERT (Agent-oriented Language for Building and Eliciting Real-Time requirements) is a formal language for requirements specification centred around the notion of agent.

Goal-oriented requirements engineering is closely related to agent-oriented requirements engineering but explicitly captures non-functional requirements such as reliability, flexibility, integrity and adaptability, by representing them as particular cases of goals (sometimes called *soft-goals*). Examples of this approach are KAOS (Knowledge Acquisition in autOmated Specification) [22], which is a formal framework focused on requirements acquisition[1], and NFR [18], which focuses on the representation of, and reasoning about, non-functional requirements.

2.5.2 Languages

Languages are used during several stages in the development of agent-based applications. In this section we briefly describe the most notables cases of agent-oriented languages for programming, communication and ontology specification.

The most commonly used general languages to build agent-based systems are Java and C++. However, from an agent-oriented perspective, these languages work at such a low level that it is difficult to implement agent features unless an additional platform or framework is used. An alternative approach to using frameworks consists in using higher level languages that implement

[1]Not to be confused with the KAoS (Knowledgeable Agent oriented System) system developed by Bradshaw.

agent concepts, known as agent-oriented programming languages, of which some examples are briefly described below.

- AGENT-0 is a language for the specification of agents and their behaviour, and is based on the agent-oriented programming paradigm proposed by Shoham [112]. AGENT-K [23], a development of AGENT-0, integrates KQML [34] (see below) into AGENT-0.
- Concurrent METATEM [39] uses an executable temporal logic to specify the intended behaviour of an agent. In this language, agents are viewed as concurrent processes that communicate by means of messages.
- AgentSpeak(L) [103] is a rule-based language with a formal operational semantics that views agents as composed of intentions, beliefs, recorded events and plan rules. It is based on the PRS [51] architecture.

In addition, some other high level languages, such as Prolog and LISP, have been used for the construction of multi-agent systems (particularly the knowledge component of agents [101]), or simulation environments and testbeds for multi-agent systems [32].

Agents use high-level discourse to communicate, so agent communication languages bypass low level aspects such as the characteristics of physical communication, and focus on the exchange of communicative acts and domain concepts. Some prominent examples of agent communication languages are outlined briefly below.

- Based on speech act theory, KQML [34] (knowledge query manipulation language) was originally developed as part of a DARPA project, but is nowadays perhaps the most widely used communication language.
- Similarly to KQML, FIPA-ACL [35] is part of the set of standards proposed by the Foundation for Intelligent Physical Agents (FIPA) and this has facilitated its popularity, particularly as part of application development frameworks. It has a formal semantics and is also based on speech act theory.

2.5.3 Modelling languages

In the context of this thesis, a *modelling language* is a language that allows us to express the planning of a system, or part of it. By itself, a modelling language does not suggest a way to design systems, but only provides a way to express the design. Modelling languages normally use, or include, graphical representations, making the models more comprehensive. Although traditional approaches to software engineering possess *de facto* modelling languages, such as the Unified Modelling Language (UML) in the object-based approach, in the agent-based approach none of the several proposed languages is clearly dominant. Below, we briefly describe one of the most cited of these languages, namely the Agent Unified Modelling Language (AUML).

2.5.3.1 AUML

Based on the number of methodologies that employ AUML [66] (for example, ADELFE [100], Tropos [9], PASSI [20] and INGENIAS [96]), we can argue that this language is one of the most popular languages for modelling agent-based systems. Perhaps its popularity results from the fact that it extends UML, a language widely used in the design of object-based systems, in a straightforward fashion, so that anyone familiar with UML should be capable of using the agent features.

AUML extends UML mainly in the sequence diagrams and in the class diagrams. These extended sequence diagrams are arguably the most used aspect of AUML. Below, we present a summary of how UML sequence diagrams have been modified to consider agent concepts [66].

- Agents, roles, and agent classes appear in the boxes at the top of the diagrams, instead of object classes and object instances. For instance, in Figure 2.3, the boxes — pointed to by marker 1 — are occupied by the *Buyer*, *Marketer* and *Seller* roles. (Roles are denoted by a slash before their name.)
- Instead of *focus of control* — as in UML — the lifelines of the diagrams contain *threads of interaction*. These threads are depicted in the diagram as narrow rectangles, such as the one pointed to by marker 2 in Figure 2.3.
- In order to express parallelism when sending messages, AND, OR and XOR connectors (applied to a set of messages) have been added, expressing that all the messages, several of them, or just one, respectively, can be sent at the same time. These connectors are represented graphically as shown in Figure 2.4, in which three messages have been depicted for each connector.
- When using FIPA-ACL [35], message arrows can be labelled with the communicative act that they represent, as in the communicative act *request* in Figure 2.3, and indicated by marker 3. Also, the number of messages sent and the number of role instances that receive the message may be added to the arrows. Examples of this can be seen in Figure 2.3, and are indicated by marker 4.
- To promote re-usability, AUML allows the definition of a protocol inside another protocol in the same sequence diagram, either by *nesting* or by *interleaving*. An example of the former appears in Figure 2.5, in which the nested protocol is denoted by means of a rounded corner box.
- By using *protocol templates*, AUML allows one unit to encompass several protocols that change only in the value of some parameters, thus promoting re-usability.

In AUML, object classes are extended to *agent* classes. An agent class contains: an agent name; a state description; actions; methods; capabilities, service description, and supported protocols;

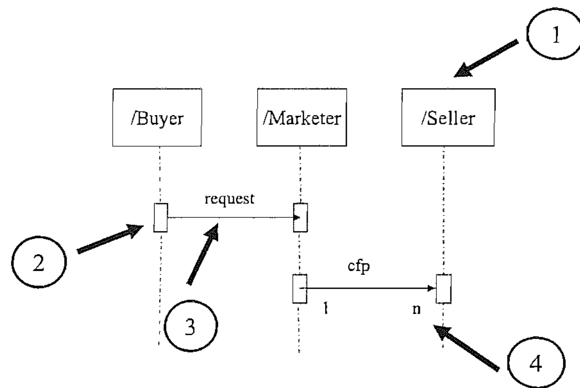

FIGURE 2.3: Example of an AUML sequence diagram

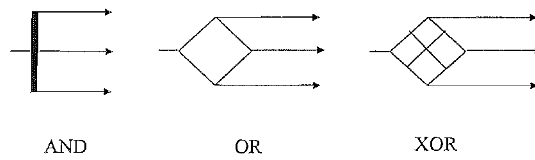

FIGURE 2.4: The AUML connectors

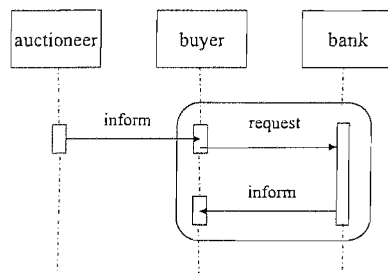

FIGURE 2.5: Example of nested protocol in AUML (after [66])

and organisations. Agent names are prefixed with the stereotype *agent* in order to differentiate them from object names. Apart from this, the instance, role or class name may be included with the name of an agent. State description in an agent class is similar to the attributes section of object classes and, in the case of BDI agents, may contain beliefs, desires, intentions and goals. Actions can be of two types in AUML: pro-active, which are those actions triggered by the agent itself; and reactive, triggered by messages received from other agents. Methods are similar to UML operations, with the addition of pre-conditions and post-conditions. Capabilities are like UML responsibilities, while service descriptions are similar to UML interfaces. The last part, organisations, is a list of groups to which an agent may belong, including the constraints for joining the group and the roles the agent plays within it.

Note that AUML is an evolving effort, subject to continuous updates and additions, whose future work considers not only modifications to diagrams but also the creation of tools to support their production, and the definition of its semantics. In spite of its benefits, and as its authors argue, AUML is not intended to be used to model all the characteristics of a multi-agent system, but it might be complemented with other languages and notations.

2.5.3.2 Others

Other modelling languages have been used in specific parts of the design of multi-agents systems. For instance, Cost et al. [21] employ *Coloured Petri Nets* for modelling agent communicative interactions, including support for concurrency. Similarly, DeLoach [85] employs *Finite State Machines* for constructing conversations in the design phase of the MaSE methodology.

2.5.4 Platforms

In the context of AOSE, we use the term *platform* to denote an infrastructure that provides facilities for the operation of a multi-agent system. Such facilities vary from platform to platform, but typically include low-level communication protocols (such as TCP/IP), agent construction, and agent management (such as registry in the platform, and white and yellow pages). In addition to this, most platforms also include support for one or more phases of the development process, analysis, design, implementation, testing and debugging. Notable examples of academic and commercial tools are: ZEUS [120], developed at British Telecom; JADE [70], developed at the University of Parma; JATLite [41], for the development of agents that communicate using the Internet; FIPA-OS [36], for the development of FIPA compliant agents; and JACK [116], oriented to BDI agents. Since it is not the intention of this thesis to go into details of implementation, we will not attempt to review all these platforms, but consider only two of them below. For more complete reviews, and comparisons of different platforms, the reader is referred to [92, 105].

2.5.4.1 JADE

JADE [70] is an evolving project developed in the Telecom Italia Labs since 1999, consisting mainly of a FIPA-compliant agent platform and tools for development of multi-agent systems, including an applications programming interface (API). JADE has a large community of users who are continually providing feedback and contributing with development and additional tools.

JADE offers facilities for the construction of individual agents, as well as for the ensemble of agents in a system. Regarding the construction of individual agents, JADE provides classes that implement the agents' basic functionality, independently of any particular architecture. Such functionality, based on the concepts of autonomy and sociability, views an agent as an *active* object (i.e., an object with its own thread of execution), able to hold multiple conversations through an asynchronous messaging protocol. In order to build an agent, developers extend the *agent* class, giving agents access to a private message queue and to facilities for processing FIPA-ACL messages. In order to implement agent tasks, JADE uses the concept of *behaviours*, which are obtained by extending the *behaviour* class provided. For their execution, behaviours are placed in a *behaviour stack list* that operates on a round non-pre-emptive scheduling policy, but more sophisticated scheduling mechanisms are also provided, such as cyclic execution and finite state machine implementations for composite behaviours.

Regarding the ensemble of agents to form a system, JADE offers a platform that complies with FIPA specifications, including components for agent management (access to the platform, white pages), directory facilitators (yellow pages), implementation of communicative and content languages (ACL and SL-0) and support for FIPA interaction protocols.

Finally, although no software tools are provided for guiding the process of developing a MAS, JADE offers comprehensive documentation of the APIs, a programmer's guide and many examples to support the process.

2.5.4.2 ZEUS

ZEUS [120] is a platform for agent-based system development originally built at British Telecom but now available as a free software project. ZEUS was developed with the purpose of providing a platform that offered information discovery (information about the agents in the system), communication, tools for ontology definition, coordination of agents, and integration of agents with legacy software.

Agents in ZEUS are formed of components that correspond to common functionalities including planning, scheduling, communication skills, coordination, and ontology support. To fit a specific application, the designer can arrange these components in different, although limited, ways.

In ZEUS, a multi-agent system is formed by using two special agents — utility agents — that carry out the tasks of system management, namely the *agent name server* and the *facilitator*.

The former functions as a white pages directory and also provides the system with a clock for synchronisation purposes, while the latter acts as a yellow pages directory.

ZEUS provides three types of facilities for achieving coordination between agents, by using protocols (based on the *contract net protocol*), by defining *roles* — such as *peer*, *subordinate*, and *superior* — that can be used to define organisational structures, and by planning.

2.5.4.3 Industry-oriented platforms

There are also several platforms for constructing multi-agent systems specifically oriented to the development of *industrial* applications, among which AdaptivEnterprise, Living Systems Technology Suite, Magenta Multi-agent Platform, and the Lost Wax Agent Framework are representatives, and briefly described below.

AdaptivEnterprise [119, 3] is a system developed by Agentis that includes a framework for developing multi-agent systems. Using this framework, complex applications can be designed and maintained graphically as simple plans, each consisting of only a few steps. By using code generation tools, agents can be automatically obtained from the graphic designs. These agents are java components capable of operating in diverse environments, and are based on a BDI architecture closely related to dMARS. AdaptivEnterprise has been used to construct applications in areas such as financial services, insurance, retail and distribution, logistics, and energy industries.

Living Systems Technology Suite (LS/TS) [118] is a set of tools developed by Whitestein Technologies that consists of a run-time platform and development tools. The run-time platform provides the middleware that adds functionality to the java run-time environment for supporting agent-based applications, and is available in personal, business and enterprise editions. The core of this run-time platform, the Core Agent Layer, defines and implements agent abstractions such as pro-activeness, goal-driven behaviour and flexible communication, and provides services such as directory services, notification services and access to the messaging infrastructure. On the other hand, the development tools include tools for design, implementation, debugging, deployment, monitoring, testing, and a methodology for systems development based on a modelling language that extends UML with agent abstractions. LS/TS is used as a basis for the creation of applications in sectors such as logistics, telecommunications and financial services.

Magenta Multi-agent Platform [2] is a java-based library of tools for developing and executing multi-agent systems. In this platform, every problem is considered as a problem of allocating resources, such as vehicles or drivers, to demands, such as transportation instructions. Agents match demands to resources through negotiating with each other. The core of the platform consists of generic components common to all applications, such as negotiation protocols, as well as components for creating the environment in which such negotiations occur, for supporting messaging mechanisms, for defining the ontology and decision making logics and process for

specific problem domains, and for data analysis. Magenta technology has been used in applications such as car production planning and scheduling, management of fleet of vehicles and planning of a complex logistics network for the transportation of heavy loads.

Similarly, the Lost Wax Agent Framework [46] is an environment for development and deployment of multi-agent systems, and provides the following facilities: support for integration to other agent environments, external systems, and external databases; an integrated security layer; CASE tools for applications design; libraries of standard protocols; and libraries of architectural templates for agent modelling. This framework has been applied to solutions in sectors such as supply chains and logistics, manufacturing, telecommunications, and financial services.

2.5.5 Methodologies

Software methodologies have proved to be successful in increasing the speed of development, in improving the quality of software, and in reducing development costs. Methodologies play such an important role that some consider the broader acceptance of agent systems to be closely tied to the availability and accessibility of adequate methodologies [89, 97, 67].

This section is devoted to reviewing the current state of agent-oriented methodologies, but it should be noted that not all the approaches considered here are methodologies in a strict sense. According to Kearney et al. [78], a software engineering methodology provides methods, guidelines, descriptions and tools for each stage in the life of a system. However, very little work involving agents can satisfy this view since most address only some parts of the system life cycle, and the tools they provide are scarce. Thus, the criteria we use as a basis for including work in this section is that it should address at least considerable aspects of the analysis and design phases, since these are the phases on which this thesis is focused. Here, we consider analysis and design as consisting of the acquisition of user requirements and their representation in some model from which an implementation that fulfils them can be built. Additionally, we focus our analysis on general-purpose methodologies, thus leaving out specific-purpose methodologies such as the methodology of Bussmann et al. [12] for manufacturing processes, and ADELFE [100] for auto-adaptable systems.

Existing agent-oriented methodologies can be classified in several ways [124, 67]. According to their origin, methodologies can be divided into those that extend object technology, those based on multi-agent systems concepts and those based on knowledge engineering techniques, considered in turn below.

Methodologies that extend the object paradigm Many of the first attempts to engineer agent-based systems extended, in some way, object technology. In fact, object technology continues to be a source of inspiration for some aspects of more recent methodologies. This seems natural since the object paradigm is, at present, the most mature and most used paradigm for

software development. In addition, there are several benefits of using object technology as a base for an agent methodology [67], as follows.

- The object-oriented and agent-oriented paradigms have several similarities since both encapsulate knowledge (data) and behaviour (methods). Also, in both paradigms communication is achieved by message passing. It has even been claimed that agents are *active objects* or *objects with an attitude* [31], although we believe that such a claim oversimplifies the characteristics agents possess (unless in *attitude* we encompass all the characteristics mentioned previously).
- Object-oriented languages have been used to implement agent programming frameworks, such as JADE [70]. Although such frameworks work at a higher level of abstraction, they still reflect the characteristics of the object paradigm.
- The types of models used by object-oriented methodologies, *static*, *dynamic* and *functional*, can be satisfactorily applied to agents. The static view deals with representing structural properties and has been used to model the inner structure of agents, and the static relationships between them, such as aggregation. The dynamic view, which describes how the elements of a system interact at run-time, is employed mainly to represent protocols of interaction between agents. Finally, the functional view, that describes the functionality of a system and how it is decomposed, is useful to represent the data flow present in agent activities.
- Some well known techniques for object-based design and analysis can be successfully extended to agent-based systems. For example, the use of *use cases* and *class-responsibility-collaboration cards* (CRC cards) are helpful in identifying agents during the analysis phase. Use cases are abstractions that decompose the functionality of a system into well identified parts, and represent the external actors that interact with the system to pursue them. A CRC card is a physical card that contains the name of an object class, its responsibilities, and the name of the classes with which it collaborates to fulfil them. Due to their simplicity, CRC cards have the benefit of forcing designers to focus on the identification of classes and collaborations, bypassing details that are irrelevant at that phase of the analysis.
- The popularity of object-oriented methodologies potentially increases the number of users of the agent-oriented methodology.

Two examples of methodologies that extend object technology are MaSE (Multiagent Systems Engineering) [24] and the KGR (Kinny, Georgeff and Rao) method [80]. MaSE is a methodology that defines two languages to model agent-based systems: the Agent Modelling Language and the Agent Definition Language. The former is a graphical language to describe the types of agents of a system and their interfaces to other agents, while the latter is based on first order predicate logic and is used to describe the internal behaviour of agents. Both languages can be

used as part of a traditional development technique or as part of a formal system synthesis. On the other hand, the KGR method divides design into two parts, one for modelling the interaction between the agents, and the other to model the internal aspects of each agent. For the latter part, KGR uses a BDI architecture. A more detailed description of these two methodologies is given in the next section.

Methodologies based on knowledge engineering techniques For some years, methodologies have existed to engineer knowledge-based systems. If we assume that all agents have a component of knowledge, then it logically follows that extensions of these methodologies could be developed to engineer agent systems. However, these extensions must address aspects usually not covered in these methodologies, such as the distributed and social aspects of agents and their goal-oriented behaviour. Representatives of this approach, based on knowledge engineering techniques, are CoMoMAS [53] and MAS-CommonKADS [68]. The latter is described in more detail in the next section.

Methodologies based on multi-agent systems concepts Methodologies based on multi-agent systems concepts do not find their inspiration from other areas of computer science, such as the object paradigm or knowledge engineering technology, but from concepts derived from multi-agent systems, which are in turn inspired by interdisciplinary areas of knowledge such as organisation theory and coordination theory. Implicit in the approach of these methodologies is the assumption that, although agent-oriented methodologies based on the object paradigm have succeeded to some extent, the full potential of the multi-agent paradigm cannot be reached by just extending the object model. Representatives of this kind of methodology are Gaia [126] (which is described in the next section), Gaia extended with additional organisational concepts [133], and SODA [99].

2.6 Representative agent-oriented methodologies

As previously noted, many methodologies have been proposed to date. Since it is impractical to describe all of them, we have selected only a few to be described in more detail as representatives of their corresponding group. We based our selection on aspects such as how much they have been referenced by the agent community, how much they have influenced other work, the amount and accessibility of their literature, and how long they have been around.

2.6.1 Gaia

Proposed by Wooldridge, Jennings and Kinny, Gaia [126] is a methodology for analysis and the first stages of design, and is applicable to problems where the number of agents is not much greater than one hundred. In Gaia, the development of a system is seen as a process of

organisational design, in which the system is formed of roles with associated responsibilities, permissions, activities and protocols. The process of modelling a system is divided into two parts, analysis and design, each of which deals with the development and refinement of the corresponding models.

The *analysis* models consist of the roles model and the interaction model. During the creation of the roles model, the main roles in the system are identified, together with their responsibilities and permissions. The roles model itself consists of a set of *role schemata*, one for each role, and each comprising the description, protocols, activities, permissions, and responsibilities of a role. Patterns of interaction between agents are modelled in the interaction model, which consists of a set of *protocol definitions*, each comprising the description, initiator, responder, inputs, outputs and processing of an interaction between two agents. The analysis phase in Gaia is iterative. First, the main roles are identified, and then, for each role, its associated protocols are documented. This leads to the refinement of the roles model, which in turn can lead to refinements of the interactions model and so on.

The Gaia *design* phase yields three models, the agent model, the services model and the acquaintance model. The agent model is a tree where leaf nodes are roles and other nodes are agent types comprising one or more roles. For example, for efficiency, a designer might decide to put together three roles in an agent type, instead of having three different agents. The services model is just a list of services of all the agents in the system, together with their inputs, outputs, pre-conditions and post-conditions. A service is defined as a single, coherent block of activity in which an agent will engage, and its characteristics are derived from the protocols model. Finally, the acquaintance model is a directed graph, where nodes are agent types and arcs are potential communication paths, so that there is an arc from node a to node b if at least one message would be sent from a to b. The acquaintance model can be derived from the roles, protocols and agent models, and it is useful to identify potential communication bottlenecks. Note that in Gaia, the design is not intended to produce an output detailed enough to be implemented on a particular platform, but it focuses on describing how the agents cooperate to achieve the system goals.

2.6.2 KGR

The KGR (Kinny, Georgeff, Rao) methodology [80] models an application from two viewpoints: the external and the internal. The external viewpoint focuses on the interactions between agents, whereas the internal viewpoint deals with the composition of each agent. Two models form the external viewpoint: the agent model and the interaction model. The agent model represents the hierarchical relationship among agent classes, while the interaction model represents the responsibilities of agents, the services they perform to achieve them, and the interactions and control relationships among agents. In order to develop these models, the following steps are established.

1. Find the main roles of the application and create a preliminary agent hierarchy.

2. For each role, find its responsibilities and the services needed to provide those responsibilities. Consider not only interaction between agents, but also interaction with the environment and human users.
3. For each service, identify interactions (performatives and content), determine control relationships between agents, and model the internal structure of each agent.
4. Refine the agent hierarchy by introducing inheritance and aggregation where appropriate, creating concrete agent classes, refining control relationships and introducing agent instances.

The internal viewpoint is based on the BDI architecture, which views an agent as having mental attitudes of belief, desire and intention representing, respectively, the information, motivation, and deliberative states of the agent [104]. According to this, agents are characterised by the events they perceive, the beliefs they hold, the goals they adopt and the plans to achieve those goals. The internal viewpoint consists of three models: the belief model, the goal model and the plan model. The first comprises information about the environment, the second deals with the goals an agent can adopt and the events it responds to, and the last regards the plans to employ to achieve the goals. Roughly speaking, the steps to model the internal viewpoint are the following.

1. For each goal, analyse the different context in which it is to be achieved. For each of these contexts, establish how the goal is to be achieved in terms of sub-goals and operations. Do the same for sub-goals.
2. Build the beliefs from the contexts and conditions that control the execution of actions and activities.
3. Iterate the above steps as the models are refined.

It is assumed that the results of this methodology are used in an infrastructure like dMARS [28], which provides additional execution mechanisms. For example, it describes how events and goals produce intentions and how intentions lead to actions.

2.6.3 MAS-CommonKADS

The MAS-CommonKADS methodology [68] extends the CommonKADS methodology, and is based not only on knowledge engineering but also on object-orientation and protocol engineering. It can be divided into three phases: conceptualisation, analysis and design. The first is an informal phase, while the analysis and design phases consist of the development of some models and textual documents. The process of development is *risk driven*, so that the components with the highest risk are tackled first, helping to reduce the risk of the project and to meet deadlines.

In the conceptualisation phase, a preliminary description of the system is produced by means of use cases, in which the interactions are graphically represented using Message Sequence Charts [107].

The analysis phase yields the following models.

- The *agent* model describes agent characteristics such as reasoning capabilities, services and goals. The methodology proposes several helpful techniques to identify agents and their characteristics.
- The *task* model describes the tasks that the agents carry out and their decomposition.
- The *expertise* model describes the knowledge needed by the agents to achieve their tasks, and is divided into domain, task, inference and problem solving knowledge.
- The *organisation* model describes the organisation in which the agents will operate, as well as the organisation of the agent society.
- The *coordination* model describes the conversations between agents, including protocols and required capabilities.
- The *communication* model describes the interactions between humans and agents.

The design phase yields only the design model, which collects the analysis models and is divided into the following three parts.

- *Application design* is concerned with the composition or decomposition of agents and the selection of a suitable architecture for each agent.
- *Architecture design* deals with the design of agent network facilities.
- *Platform design* concerns the selection of a development platform for each agent architecture.

2.6.4 MaSE

MaSE (Multi-agent Systems Engineering) is an agent-oriented methodology based on object techniques that cover most phases of the development cycle: analysis, design and implementation. There is also a CASE tool (agentTool) that supports all the phases of the methodology. The main philosophy of MaSE is the construction of graphical models, each of which is built according to guides provided by the methodology. The construction of a system begins with a set of initial requirements and ends with a specification that is independent of any agent architecture or platform. The development of a system is divided in MaSE into analysis and design, each of which consists of several phases, as described below. However, it must be noted that the

phases do not follow a plain sequential order, but iterations are allowed and encouraged in order to gradually refine the design.

The analysis encompasses the following three main activities.

- Identification of goals and their decomposition into sub-goals.
- Identification of use cases and creation of sequence diagrams to identify roles and their interaction.
- Transformation of goals into roles.

On the other hand, the design consists of the following four phases.

- Assignment of roles to agent classes (which are similar to object classes but incorporate agent features), and identification of conversations (patterns of interaction between the roles).
- Detailing the conversations.
- Definition of the internal agent structure.
- Definition of the final system structure.

MaSE is a good example of a methodology that is continually evolving. From the methodology described in [24] to the one described in [85], MaSE has gained in comprehensiveness, coverage of phases and supporting tools. Also, the additions presented in [25] have made MaSE useful for developing open systems. Other additions like verification of conversations and incorporation of an ontology model have made MaSE one of the most complete current methodologies.

2.6.5 Tropos

Tropos [15, 82, 81] is an evolving project being developed by researchers from several universities. Its main purpose is to provide help during software development of systems in changing environments. To this end, techniques, tools and a methodology are provided. The methodology, which is also named *Tropos*, has the following characteristics.

- Tropos covers a wide range of development phases: early requirements, late requirements, architectural design, detailed design and implementation.
- All the phases are *requirements-driven*, which means that the modelling concepts used in the early requirements phase are consistently used along the other phases. This supposedly reduces the gap between the system and its environment.
- Tropos is based on organisational concepts.

- Tropos incorporates the use of organisational patterns of two types: *architectural styles* and *social patterns*. A style is used as a system architecture (a small, intellectually manageable model of system structure, which describes how system components work together) during the architectural design. On the other hand, social patterns are used to represent how a specific role goal is fulfilled by agents.

- In contrast to other agent-oriented software engineering projects, the literature and information about Tropos is vast [9, 15, 82, 81, 102].

- Tropos uses (and in some cases extends) well-established tools and frameworks, for example, i* [130] for organisational modelling, AUML (see Section 2.5.3.1) as a graphical modelling language, and JACK [116] as an implementation platform. i* is a modelling framework built around the notion of agents with intentional properties such as goals and commitments, and JACK is a commercial agent platform for BDI agents.

2.6.6 INGENIAS

INGENIAS [55] is a project developed at the Universidad Complutense of Madrid that consists of a software methodology as well as set of tools. According to the purpose of this section, we will focus our description only on the methodology. INGENIAS views a system as divided into five complementary viewpoints, namely organisation, agents, tasks/goals, interactions and environment. The organisation viewpoint describes a structure in which the elements of the system are arranged. These elements include agents, resources, goals, tasks, groups (of agents, roles, resources or applications), workflows (associations among tasks and the information for their execution) and social relationships (restrictions on the interaction between entities).

The agents viewpoint deals with the internal functionality of the agents and is divided into three components: mental state, mental state manager and mental state processor. The first component is formed of all the information an agent needs to take decisions, the mental state manager performs operations on the mental entities (such as creating, destroying and modifying them), and the mental state processor determines how the mental state evolves based on, for example, rules and plans.

The tasks/goals viewpoint addresses the decomposition of goals and tasks, including the reasons why they should be performed and the consequences of their execution.

The interactions viewpoint deals with the requests or exchange of information between agents and between agents and human users, identifying the participants of the interaction, the interaction units (messages and speech acts), protocols involved, context (goal pursued and mental state of the participants) and coordination mechanisms used.

Last, the environment viewpoint defines the entities with which the system interacts. These entities can take the form of resources (entities that do not provide an API), applications (entities for which an API is provided), and other agents outside of the system.

On the other hand, the main idea behind the methodological process of INGENIAS [54] is the adaptation of the Unified Process [69] (RUP) to employ agents rather than objects. For this, associations are proposed among the elements of RUP and those defined in the viewpoints cited above: classes are matched with agents, organisations with architectures, groups with subsystems, interactions with scenarios, and roles, tasks and workflows with functionality. Based on these associations, the resulting activities of the process are obtained. First, the main workflows (analysis and design) are divided into three phases: inception, elaboration and construction. During the analysis, organisation models are produced to obtain the general architecture of the system, and these models are refined to identify common goals and relevant individual tasks. During the design, more detail is added by defining workflows and refining the agents' mental states. The previous activities are first carried out for the most significant use cases (in elaboration phase) and then for the rest of them (in the construction phase). Regarding the implementation, two options are possible: manual implementation using the specification provided by the models, or automatic code generation using a provided tool, which generates code in Java, Java Expert Systems Shell, April and Prolog. Finally, the testing workflow is no different from conventional software testing.

2.6.7 Others

There have been several approaches to formally specifying and implementing agent systems. Different formalisms have been employed to this end; for example, Luck et al. [90] provide an outline for a possible methodology for the development of agent systems using the Z language (hereafter called the Formal Agent Framework). Such a methodology is based on a formal framework, which uses a hierarchy of entities (objects, agents and autonomous agents) to model an application domain. Roughly, the process to develop an application is initiated by identifying the entities, their purpose, and their control relationships. This results as a classification of the entities in objects, agents or autonomous agents. The process continues with the design of methods for behaviour and the control relationships, and ends by identifying structural similarities in the entities in order to exploit them. The final product of this process is a specification whose implementation by means of objects is straightforward. One restriction in this framework is that it is difficult to express time properties about the behaviour of entities, due to the fact that Z has no notion of time. Future work could tackle this restriction by using another formalism, either alone or in combination with Z [90].

MESSAGE [78, 13] (Methodology for engineering systems of software agents) is a methodology designed by the EURESCOM consortium, mainly oriented towards the telecommunications domain. MESSAGE extends some elements commonly found in object-oriented methodologies — particularly RUP [117] — such as the use of UML [43] as the modelling notation, and a methodological process based on iterations and increments.

SODA [99] is a methodology based on organisational concepts that covers only the phases of analysis and interaction design. There are three characteristics that distinguish SODA from other

similar methodologies: the use of *task*, instead of *role*, as the primitive concept; the importance given to the environment of a system during its modelling; and the use of interaction rules to enforce the accomplishment of social tasks.

2.7 Evaluation of methodologies

In this section we present an evaluation of the agent-oriented methodologies reviewed above. Such an evaluation is important for the identification not only of the weak points of each individual methodology, but also of the drawbacks of the overall agent-based approach. Although several evaluations exist in the literature [111, 5], the evaluation presented in this section is different in that it is oriented to determine the suitability of the methodologies to be used in commercial and industrial environments by software engineers (in contrast to academic and research communities of agent specialists).

We have limited our evaluation to the available literature for each methodology in the previous section. However, we are aware that some of these methodologies are evolving projects, and so are prone to updates. Also, for the purpose of the evaluation, we define the meaning of the terms used and make an effort to match these definitions to those used by the authors of the methodologies.

2.7.1 Development process

We first evaluate the coverage of the development process. To this end, we divide the development process of an agent-based application into the following phases, with their attached meanings.

Requirements The collection and organisation of the requirements of the system.

Analysis The understanding of the system, its main components, and its environment

Structural design The specification of how the system fulfils its requirements, focusing on the interactions between the agents.

Architecture design The specification of how the system fulfils its requirements, focusing on the internal composition of each agent in the system.

Implementation The activities involved in the construction of an executable system from the specification of the design phases.

Table 2.1 shows which of these phases are covered by each methodology. As can be seen from the table, most methodologies fail to consider all the development phases.

Gaia describes in appropriate detail the phases that it covers, but the application of the methodology relies on the use of artefacts that are not sufficiently described, such as the organisational patterns and the language for describing structures. Also, the fact that Gaia covers only two development phases limits its use in real-world applications, but the neutrality of these phases makes Gaia attractive for extension, not only in terms of increasing the phases covered, but also in enriching the process with the addition of iterations and increments.

KGR strongly relies on the object approach for describing the design of a system. It is also strongly oriented towards the use of BDI architectures, particularly in the architecture design phase, and although architecture-dependence is not regarded as a desirable characteristic, the use of BDI architectures with KGR have proven to be successful in real-world applications. Additionally, some parts of the methodological process are not described with enough detail, and iterative design is not explicitly considered.

The way in which the process of MAS-CommonKADS is specified presents some drawbacks. First, the process is described as an extension of CommonKADS, which is natural but assumes that the practitioner is familiar with CommonKADS, which is not usually the case for a *typical* software engineer. Second, the process is centred around the description of the models, but less emphasis is placed on how to obtain the relevant information, and the order in which to develop the models, including iterations. In addition, the architecture design is limited to the extent of selecting an appropriate architecture for each agent, but does not cope with obtaining a detailed specification of an agent.

Tropos covers all the development phases considered in our review, although in some cases it is oriented towards a specific architecture or platform. The architecture design considers only one agent architecture (which happens to be a purpose-built architecture based on mentalistic notions such as goals and plans), and the implementation relies on the JACK platform [116]. The organisational patterns (called *architectural styles* in Tropos terminology) are useful to describe the architecture of a system at a high level, but further refinement is needed to apply them to specific applications.

INGENIAS has good coverage of phases and it is said to have an iterative-incremental process, although no details are provided. Also, INGENIAS is highly architecture and platform independent, but at the cost of forcing the developer to build from scratch the description of the system by means of diagrams.

2.7.2 Facilitators

We have also considered the elements of the methodology oriented to facilitate its application to real world problems, hereafter called *facilitators*. These elements are diverse, but have have in common that without them any practical design would tend to be highly time-consuming and its results unreliable. In particular, in this evaluation we consider the following facilitators.

Methodology	Req.	Anal.	Struct. Des.	Arch. Des.	Imp.
Gaia	×	√	√	×	×
KGR	×	√	√	√	×
MAS-CommonKADS	√	√	√	√	×
MaSE	×	√	√	√	√
Tropos	√	√	√	√	√
INGENIAS	×	√	√	√	√

TABLE 2.1: Covered phases

Graphical models The presence of adequate graphical models to describe the properties of the system and specify its implementation.

Techniques Procedures to guide the derivation of relevant information and the accomplishment of the activities of the process.

Tools Mechanisms that accelerates the development, such as automatic code generation and reusable libraries (*patterns*).

Graphical tools The existence of graphical software tools, as integrated development interfaces (IDEs), to support one or more of the development phases.

Table 2.2 shows the existence of the different types of facilitators in each methodology. In the table, a facilitator is said to be present in the methodology only if plays a relevant role in it. For example, the *techniques* column is ticked only if the methodology provides techniques for a satisfactory proportion of the activities.

It can be observed from the table that only a few methodologies provide all the types of facilitators considered, and their weakest point is the lack of techniques and tools. Regarding graphical models, all the methodologies include them in some form, although with drawbacks in some cases. Gaia, for example, uses graphical models to describe components of the system but not the relationships between them. For instance, the *role model* shows each role of the system, but does not show their relationship of interaction or dependency. In addition, most of the graphical models of these methodologies are based on those of the agent paradigm, and none of the methodologies considers the use of the Agent Unified Modelling Language (AUML).

The inclusion of techniques as part of the process is important for the purpose of smoothing the learning curve for novice practitioners. However, only a few methodologies provide detailed and complete guidance about how to obtain the information required by the models, and the way in which this information is related. This could be explained partially by the lack of extensive documentation that most methodologies suffer, and is exemplified by the MAS-CommonKADS methodology.

Methodology	Graphical models	Techniques	Tools	IDE
Gaia	√	×	×	×
KGR	√	√	×	×
MAS-CommonKADS	√	×	×	×
MaSE	√	√	√	√
Tropos	√	√	√	×
INGENIAS	√	√	√	√

TABLE 2.2: Facilitators

Although tools have been recognised as an important factor in spreading the use of agent-oriented methodologies, they are not included in most methodologies. One of the reasons for this is that agent-oriented methodologies are relatively recent and the development of tools requires additional effort and time. Tools are present in these methodologies mainly in the form of reusable patterns and code generation modules. Reusable patterns play an important role in the Tropos methodology, although they are situated at such a high level that they are applicable only for specific types of applications, for example electronic commerce. Code generation has been used as a means to avoid platform and language dependence.

Similarly, IDEs help to substantially reduce development time, but surprisingly only a few methodologies include them. Current IDEs are being used mainly for building diagrams and code generation, but can also be used for checking completeness and consistency, guiding the development process, and as a forefront for other tools such as compilers and debuggers.

2.8 Drawbacks in current methodologies

It is desirable to have standard criteria to evaluate methodologies. Shehory and Sturm [111] established some criteria to evaluate modelling techniques for agent-based systems, divided in two parts: software engineering evaluation and agent-based system characteristics. The former encompasses desirable characteristics of a modelling technique from the viewpoint of software engineering, but deals mostly with how simple to use and how powerful the technique is. The following are qualities assessed in the software engineering aspect of an evaluation.

Preciseness How unambiguous the semantics of the models is.

Accessibility How comprehensible the technique is for beginners and experts.

Expressiveness How applicable to multiple domains the technique is.

Modularity How stable to the introduction of new requirements the resulted models are.

Complexity Management How easy it is to work at different levels of abstraction.

Executability The quantity and quality of tools to support prototyping or simulation of at least one aspect of the methodology.

Refinability How easy it is to obtain an implementation from the design specification.

Analysability The quantity and quality of tools to check consistency and coverage.

Openness The applicability of the methodology to different architectures, infrastructures and programming tools.

The agent-based system characteristics part of the criteria focus on the evaluation of the methodology to support agent-oriented features. The characteristics included are the following.

Autonomy How well supported the representation of the self-control property is.

Complexity How well supported the complexity management is.

Adaptability The support for creating agents adaptable to dynamic changes in the environment.

Concurrency How well represented concurrent agent operation is.

Distribution How good the methodology is for representing the distribution over a network of the multi-agent system components.

Communication richness What support it provides to express the communication richness required to model agent interactions.

Shehory and Sturm applied these criteria to the evaluation of some methodologies [111, 110], including Gaia [126], the Formal Agent Framework [90] and AUML [66]. Their conclusions show that these methodologies adequately address agent characteristics like autonomy, complexity and adaptability. However, they are poor in addressing agent characteristics like concurrency, distribution and communication richness. In spite of this, they see the software engineering issues as the weakest point in the evaluated techniques.

There are several surveys of agent-oriented methodologies [5, 67, 111, 124], many of which also attempt to provide an evaluation of the reviewed methodologies. One can note that there is some consensus about the weak points of current methodologies. In general, the weak points refer to the lack of software engineering features, like complete coverage of the development cycle.

In our evaluation, we identified drawbacks in current methodologies that complement those found in other evaluations. These drawbacks can be grouped into four categories: supported phases (coverage of the whole development cycle), agent architecture (support for the internal design of agents), interactions in open systems (flexible modelling of agent interactions), and iterative development (development by successive executable deliveries). The drawbacks we identified are presented below, together with some possible ways to overcome them.

2.8.1 Supported phases

As was discussed in Section 2.7.1, current methodologies are not complete or detailed enough. Even if we restrict our attention only to the analysis and design phases of the development cycle, very few methodologies cover all the corresponding activities. For example, SODA and Gaia do not address intra-agent issues, some of the MAS-CommonKADS models can be the subject of further improvement [68], and MESSAGE is neither complete nor mature [78].

2.8.2 Agent architectures

It is inevitable that one must deal with specific agent architectures during the design of individual agents. We have identified the following three different approaches taken by current methodologies:

1. For some methodologies, such as Gaia [126], agent architectures are considered to be out of scope since they were designed only for analysis and high-level design.
2. Some methodologies are tied to a specific architecture; for example KGR [80] is tied to the BDI architecture.
3. Other methodologies are tied to a specific architecture but claim that the same design principles can be applied for other architectures, or that other architectures can be adapted to the architecture considered. For example, MESSAGE [78] uses a generic layer architecture as a template from which concrete architectures can be instantiated, depending on the specific characteristics of the agent in question.

Thus, few current methodologies satisfactorily incorporates at least the most popular agent architectures.

2.8.3 Interactions in open systems

Interactions are a key issue when modelling multi-agent systems, since agents achieve their goals by interacting with other agents. In the case of open systems, the interaction mechanisms should be flexible enough to allow new agents to be incorporated into the system (since although conversation protocols are naturally open, modelling *complete* agent interactions requires particular considerations, since no assumptions can be made about the identity and composition of agents). It seems that INGENIAS (one of the most mature methodologies from the software engineering viewpoint) neither facilitates nor prevents the development of open systems. It would be helpful to adapt or extend INGENIAS to explicitly consider this kind of system. In particular, enriching the analysis and upper design phases of INGENIAS with the organisational abstractions recommended by Zambonelli et al. [133] could lead to a methodology suitable for tackling open systems.

Although addressing the *semantic* aspect of interactions is outside the scope of this thesis, it is worth mentioning the approach towards this direction presented in [76], in which Johnson et al. describe a mathematical model for agent interaction. This model is based on the mathematical notion of *category*, which is a mathematical construct consisting of two sets and a system of *combination rules*. The first set (the objects) is largely a placeholder, while the second set (the morphisms) consists of a collection of arrows from one object (the tail) to another (the head). The system of combination rules consists in obtaining a larger arrow from two arrows, where the head of one is the tail of the other. The model so obtained represents formally the utterances and commitments in agent dialogues, and the relationships between them, and can be used for modelling communication languages such as FIPA-ACL, and protocols such as an English Auction.

Another relevant approach towards interaction modelling appears in [60], where the WSCI (Web Service Choreography Interface) language is presented. WSCI is an XML-based language, proposed by the W3C group, for the description of interactions between web servers, but several of its features are also applicable for other types of interactions. The features of an interaction that can be described by means of WSCI include: the order in which the messages can be sent or received in a given message exchange; the messages that form a *transaction*; exception handling; thread management; and alternatives based on run-time values. It is planned that WSCI be used together with other standards to achieve complete semantic and technical interoperability of web services.

2.8.4 Iterative development

One of the best practices adopted in successful software engineering methodologies is that of iterative development [43, 117], which encourages *vertical* development, in contrast to the classic waterfall model. Roughly, the practice consists in dividing the development of a system into iterations, each of which delivers an executable that upgrades the functionality of the previous iteration, and consists of analysis, design and implementation phases. This approach has several advantages, one of the most important being that it facilitates user feedback, thus avoiding developing the wrong system. Other advantages are that it helps the development stay on schedule and makes it easier to accommodate tactical changes [117]. None of the reviewed methodologies satisfactorily applies this best practice, but this has been identified as important [78].

2.9 Conclusions

Despite the existing work in agent-oriented software engineering, there remain outstanding problems. In particular, it is difficult to produce high quality tools because of the gap between theory and practice. Fisher et al. [40] point out that, for instance, many AI theories of knowledge representation are never *used* because they have not been *designed* for practical use. They

suggest that such a gap can be filled by the convergence of theories at a lower level and programming languages at a higher level, or by producing methodologies that act as a bridge. In this thesis we focus on the latter approach by attempting to complete the bridge; more precisely, we refine some of the elements found in current methodologies to make them more comprehensive and easier to understand for a broader public.

For the purpose of this chapter we have assumed a rather general concept of agenthood. However, for the rest of the thesis we need to be more precise about what an agent means, since it strongly determines the orientation and scope of the work presented in the next chapters. In the rest of the thesis we assume, therefore, that an *agent* is a software program with its own thread of execution that can exhibit run-time properties of autonomy, social ability, reactivity and pro-activeness, and that some of the several modules it encompasses can have their own thread of execution. Whether or not an agent exhibits any of these properties is determined by its design, and not by the limitations of the platforms on which it is implemented.

Based on what we presented in this chapter, we can summarise the state of agent-oriented software engineering as solid and promising but still embryonic. AOSE is solid because it is founded on well studied concepts and principles, many of which existed even before the agent approach and are the result of years of experience and refinement. AOSE is promising because there exists analytical and practical evidence showing that it is a valuable tool for developing the systems that today's complex applications require. However, from a software engineering perspective, AOSE is embryonic in the sense that it does not incorporate practices that, for several years, have proven to be useful in engineering software systems.

To overcome these limitations, in this thesis we address the issues involved in taking agent-oriented software methodologies to a point where they can be effectively applied to the development of open systems. To do so, we consider the two main aspects in which the development process is divided, namely the macro-level (design of the interactions between agents) and the micro-level (design of the internal composition of agents), as well as the process itself. Our approach consists of incorporating a set of specific software practices to the development process of agent-oriented methodologies. This set of software practices tackle precisely those areas identified as drawbacks in our evaluation, and consist basically of the use of software patterns and an incremental iterative approach for the methodological process. However, to accomplish these tasks, some other problems had to be solved first, as is explained in the following chapters.

Finally, it is important to make clear that this thesis focuses on the analysis and design phases of the multi-agent systems lifecycle, which form, arguably, the core activities of the development process. However, the lifecycle of multi-agent systems, and software systems in general, covers many other activities, which are only briefly described below. Note that this list of activities is neither exhaustive nor definitive, since the phases can be arranged and presented in different forms, and with different names.

- As was stated in Section 2.5, the *requirements engineering* phase consists of eliciting, modelling and analysing the functional and non-functional capabilities of a system, and is performed before the analysis phase.
- The *implementation* phase follows the design and consists in the creation of an executable version of the system, including the creation of source code, binary files and scripts.
- During the *evaluation*, a system is evaluated to determine if all the functional and non-functional requirements have been implemented.
- In the *testing* phase, the implementation of a system is tested for the existence of errors. This is done by feeding the system with pre-established inputs and comparing expected outcomes to actual outcomes.
- The *validation* of a system consists of determining if its functionality correctly implements its requirements; that is, if the *right* system has been developed.
- The *deployment* phase encompasses the activities involved in making a system available for use, including installation of executables and preparation of user manuals.
- The *maintenance* phase deals with modifying the system to accommodate changes to the original requirements.
- Finally, the *upgrade* phase consists of extending a system to incorporate new requirements, as well as managing the replacement, or coexistence, of different versions of the system.

Chapter 3

Modelling organisational structures

3.1 Introduction

3.1.1 Interactions and organisations

As was briefly discussed in Chapter 1, multi-agent systems provide an appropriate means for the development of complex systems as a result of their ability to naturally and effectively represent multiple loci of control, distinct and differing perspectives, and natural decentralisation and distribution. This is becoming increasingly important in emerging computational systems which are characterised by dynamism and openness, or where systems are modelled as being composed of multiple interacting and independent entities or agents.

In this view, *interaction* between agents is a key characteristic of multi-agent systems. Yet this changes the complexity of the system, making it much more difficult to manage than traditional monolithic systems. Thus, in trying to ensure that the operation of such a distributed system, in which the components interact, is effective, we need to develop appropriate control structures and mechanisms to facilitate coherence and an overall integration of these components.

It was briefly discussed earlier that the use of *organisations* has been proposed as a means of modelling and managing these interactions. Organisations are systems composed of entities (usually people), positions (or roles) and resources. In human organisations, people play roles and are in charge of executing a series of tasks. To carry out these tasks, roles interact with other roles and use resources. These interactions give rise to a network of communication paths, or the organisation *topology*, and to control relationships that determine the type of authority of some roles over others, providing a *control regime* for the organisation. Human organisations also involve the use of explicit or implicit *rules* for regulating the way in which people, roles, interactions and resources can be combined in order to exhibit coherent behaviour, and obtain meaningful results.

A primary school is a simple example of human organisation. The main roles of this organisation are the headteacher, the teacher and the student, each with clearly defined tasks; for example, evaluation is a task of the teacher role, which in turn requires the teacher to interact with the students. In a primary school the resources are varied, and include books, blackboards, classrooms, and so on. The organisation topology of a primary school resembles the form of a tree, with the headteacher situated at the root, the teachers at the branches and the students occupying the leaves. The control regime is formed by well-defined authority relationships: the students are subordinated to the teachers, and the teachers are subordinated to the headteacher. There are also many rules of organisation in a primary school, such as attendance rules and schedules.

3.1.2 Organisations in multi-agent systems

Organisations are used in agent-based software engineering as an abstraction for analysing and designing systems. Specifically, organisations are used during the design to structure multi-agent systems, since they provide a general structure in which other design components — such as roles, agents, interactions, tasks and environmental entities — can be accommodated, in a similar way as *software architectures* are used in the development of object-based systems. For example, in the Rational Unified Process methodology [117], an architecture encompasses decisions about the structure of the system, the elements that form the structure (e.g. subsystems), the interfaces of these elements, the way these elements collaborate (behaviour), and the composition of the structural and behavioural elements into subsystems. Such an architecture is used for understanding the system, organising its development and promoting reuse.

Similarly to such software architectures, there are several types of organisation. In fact, a different organisation can be obtained by varying the roles, the tasks associated with them or the way they interact. In practice, however, organisations are grouped by their topology and control regime; for example, the term *hierarchy* refers to any organisation that resembles the organisation commonly found in most businesses, regardless of its number of roles and levels of control. Each of these groups possesses different properties; for example, because of the centralised control in the apex, hierarchies are poor at reliability, since failures in the apex may produce serious consequences for the whole system. In addition, not all types of organisation are adequate for a specific application; for instance, hierarchical organisations assume that all the roles (except the apex) are willing to be subordinated to other roles, which is not the case in peer-to-peer systems.

At this point, it is important to note that this thesis deals with *static* organisational structures, this is organisational structures whose properties do not change with time. *Dynamic* organisations might require different models, particularly different forms of representation.

An example of how organisations are used to model multi-agent systems is presented in [134], in which a hypothetical system is described to automate the management of a conference. In this conference management system, a call for papers is sent and the submitted papers are received. The committee then distributes the papers for review, collects the reviews and selects the best

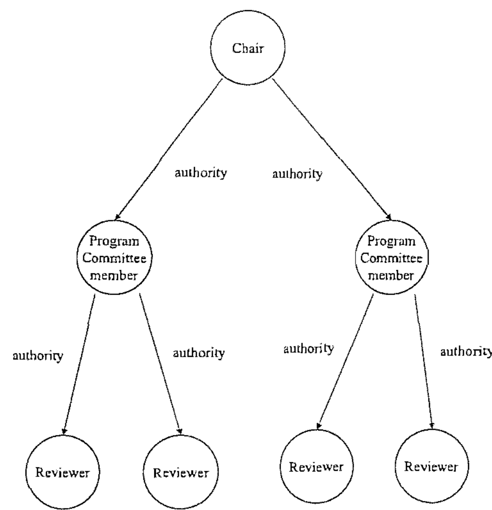

FIGURE 3.1: Organisation of the Conference Management System

papers. After an analysis of the characteristics of the system, it becomes clear that a suitable way of organising the system is by means of a three-level hierarchy in which the first level or apex is occupied by the program committee chair, the roles of the second level by other members of the program committee, and the roles of the third level by the reviewers. This hierarchy is depicted in Figure 3.1, in which roles are represented by circles and the control regime by labelled directed arrows (which in this case have the same label). According to this structure, each program committee member coordinates the activities of his corresponding reviewers, based on an *authority* relationship. Similarly, the program committee chair coordinates and supervises the activities of the program committee members. The organisation also includes restrictions on elements of the structure, such as that no agent cannot be the author and the reviewer of the same paper.

3.1.3 Organisation type selection

The *selection* of an appropriate organisation type for a given application is important since it has been shown that the type of organisation determines the performance of the system [14], and its characteristics in terms of adaptability, reliability and modularity [82]. However, selection is not a trivial task, for several reasons. First, there are, at least in principle, a large number of possible combinations of topologies, control regimes and organisational rules — that is of organisations — that can fit a specific system. Such a large number of possibilities tends to be confusing, especially for novice software development practitioners or teams. Second, the different parts of

an organisation tend to be related in a complicated way, so that giving priority to one may cause undesirable effects in others. For example, in the Conference Management System mentioned above, the coordination complexity can be reduced by replacing the three-level hierarchy by a two-level hierarchy by combining the roles *committee member* and *reviewer* in just one role. However, this increases the processing complexity of the role, making it vulnerable to delays and to failures due to a potential lack of resources, resulting in a decrease of efficiency and reliability of the organisation.

Without providing structured techniques, it is difficult to achieve the best combination of the different components of an organisation. In order to facilitate this selection of the most appropriate organisation to use for a particular application, we advocate the combined use of a suitable methodological process and adequate design tools. Within this approach, the role of the process is twofold: to obtain the necessary elements for the selection decision to be taken, and to indicate the precise stage of the development at which organisation selection must take place. The role of the tools is to provide a repository of solutions (a catalogue) from which to select an organisation, as well as a procedure to guide the selection.

3.1.4 Organisational patterns

In relation to the processes, some methodologies already exist that guide the identification of the elements for organisation selection [134, 25]. However, there are no design tools targeted specifically at supporting this selection of an organisation. To address this omission, we could consider the development of, for example, a library, encompassing the description of particular types of organisation, the situations in which their use is recommended, and their benefits and drawbacks. Such a library provides multiple benefits in the development of multi-agent systems. First, it reduces the learning curve present in the introduction of any new technology. Second, it reduces development time, since it is faster to select an organisation from the library than to construct one from scratch. Third, it promotes standards and the appearance of tools. Finally, it facilitates communication between developers, and serves as a reference and documentation.

In order to create such a library, we must deal with the description of organisations themselves, which involves the description of their topology, organisational structure and organisational rules. However, as explained below, no satisfactory form of describing organisational rules exists, so we first need to construct suitable forms of describing them. Similarly, we need to construct appropriate forms for describing organisational structures, since existing organisational models (such as [122, 58, 123]) do not explicitly consider the description of both topologies and control regimes, as well as their relationship.

The practical solution to the development of such a library is the construction of a catalogue of different organisational types, in which each such type would be a general form of organisation, or a *pattern* [48]. Basically, patterns are general solutions to recurrent problems, and can be easily specialised to provide a detailed solution for a specific situation. Patterns are suitable for

our purposes since they describe general solutions that can be adapted to particular situations, and have the benefit that they are well known to software practitioners.

In relation to this catalogue, however, the number of patterns in the catalogue must be kept manageably small to avoid a transfer of complexity from organisation construction to organisation selection. Although the number of possible organisations is infinite (one for each possible combination of topology, control regime, and set of organisational rules), they can be grouped by taking into account two considerations. First, organisational rules can be separated into domain-dependent rules and domain-independent rules, so that several organisations can be grouped if only domain-independent rules are considered. Second, the most effective reduction comes from grouping different organisations with similar characteristics into families, also called *paradigms*. For example, the *hierarchies* family encompasses single-level and multiple-level hierarchies, as well as *strict* and *non-strict* hierarchies (a hierarchy is strict if roles other than the apex communicate *only* with their superior). In addition to this form of reduction, it has been argued that that a relatively small number of families might be enough to cover most practical situations [134]. In fact, several such families have already been identified and their domain-independence verified by the number of different areas in which they have been applied [45, 64].

3.1.5 Overview

In this chapter, therefore, we present a framework in which organisational patterns can be developed. This framework includes the languages for describing the components of the patterns, a layout to specify the patterns themselves, and three examples of patterns. With this aim, the rest of the chapter is organised as follows. Firstly, in Section 3.2 we review the methodological context in which the catalogue of patterns is used. Then, in Section 3.3 we present a model for expressing organisational rules. Similarly, in Section 3.4 we present a model for describing organisational structures, including topologies and control regimes. Based on these models, the layout for patterns specification is presented in Section 3.5, and the catalogue of patterns itself in Section 3.6. Finally, our conclusions, and the benefits and limits of this work are presented in Section 3.7.

3.2 Gaia as the basis for the methodological approach

Patterns are of limited use if they are not part of a methodological process. Such a process provides the context in which the patterns are used, which includes the selection of a specific pattern, how and when to use it, as well as what is expected from it. Among all the existing processes based on organisational concepts, we adopt Gaia as the basis for the methodological context of our catalogue of patterns, because it is well known and already considers the use of organisational patterns in the design of a system. Thus, with the aim of providing the reader with the context in which patterns are used, a review of the Gaia methodology is presented below.

As described in Chapter 2, Gaia is a general-purpose methodology for the development of multi-agent systems, and was first described in [126], but has evolved significantly in several aspects up to the version presented in [134], on which this review is based. Gaia is one of the first agent-oriented methodologies that is not based directly on the object paradigm nor on knowledge engineering concepts. In contrast to these approaches, the development of a system in Gaia is based on the *organisational metaphor*. Basically, Gaia consists of a process and a series of preliminary and definitive models constructed in accordance to that process. In the following, we first present the models and then the methodological process.

3.2.1 The main models of Gaia

In this section we describe the main models in Gaia, which are: the *role* model, the *interaction* model and the model of *organisational rules*[1]. In addition, the role and interaction models can be divided into preliminary and final models, according to the information represented and to the phase of development in which they are built (more on this in Section 3.2.2). The *agent* and the *service* models complete the set of models used in Gaia (these two models are described in Section 3.2.2.3). Figure 3.2 shows the models used in Gaia, as well as their dependency relationships. For example, the role model is fed with information from the preliminary role model and provides information to construct the agent model. Although not really a model — which is indicated in the figure by a dotted box — the organisational structure has also been included with the aim of showing its dependence on the other models. In this way, the arrows entering this box denote information used to determine the organisational structure of the system, whereas the arrows leaving point to those models that use the organisational structure for their completion. However, the details about organisational structure will not be covered here, but in Section 3.2.2.2.

3.2.1.1 Role model

The role model encompasses all the roles in the system, which represents well defined positions in the organisation, and the behaviour expected from them. Roles are characterised by a set of features defining their nature and activity as shown in Table 3.1: the *name* identifies the role and reflects its main purpose; the *description* provides a brief textual description of the role; the *protocols* describe interactions with other roles; the *activities* detail those computations that the role performs without interacting with other roles; the *responsibilities* express the functionality of the role (divided into two parts: *liveness properties* and *safety properties*, which relate to states of affairs that a role must bring about, and the conditions whose compliance the role must ensure, respectively); and the *permissions* identify both the resources that the role needs in order to fulfil its responsibilities and its rights of access to use them. The characterisation of a role is

[1]Zambonelli et al. , the authors of Gaia, do not refer explicitly to the set of organisational rules as a *model*. However, we do here for clarity of exposition.

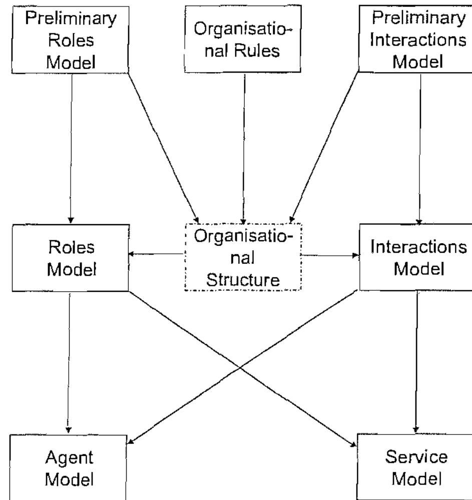

FIGURE 3.2: The models of the Gaia methodology (modified from [134])

Characteristic	Sub-characteristic	Meaning
name		identifier
description		brief description
protocols		list of protocols
activities		list of activities
responsibilities		
	liveness properties	states to bring about
	safety properties	conditions to ensure
permissions		resources and rights of access

TABLE 3.1: Characterisation of roles in Gaia

depicted graphically by means of a *role schema*, an example of which is shown in Figure 3.3. As can be observed in the figure, each box in the schema corresponds to one section of the role definition, and the names of activities have been underlined to distinguish them from names of protocols. Additionally, the responsibilities have been expressed in a purpose-built language that includes operators to represent sequence (.), alternatives (|) and indefinite repetition (^{w}).

3.2.1.2 Interaction model

The interaction model comprises all the interactions between the roles in the system. Interactions are characterised by means of *protocol definitions*, which consist of the following features:

Role Schema:	$Filter_i$
Description:	Performs the process corresponding to stage i on the input data
Protocols and Activities:	$\underline{ProcessData_i}$, GetInput, SupplyOutput, $\underline{SenseFlows}$, $\underline{ChangeFlow}$
Permissions:	changes *Data*, $flow_i$, *agreedFlow* reads $flow_j$

Responsibilities:

Liveness:

$Filter_i = (Process \mid AdjustFlow)^{\omega}$

$Process = GetInput.\underline{ProcessData_i}.SupplyOutput$

$AdjustFlow = \underline{SenseFlows} \mid \underline{ChangeFlow}$

Safety:

- true

FIGURE 3.3: Example of a role schema

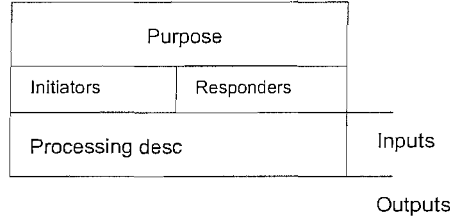

FIGURE 3.4: A generic protocol definition

a *purpose* that provides a brief textual description of the interaction; a list of *initiators* that enumerates the roles that start the interaction (usually a single element); a list of *responders* that enumerates the roles involved in the interaction, apart from the initiators; a list of *inputs* and *outputs* that provides the information required or produced during the interaction; and a brief textual *description* that outlines the processing performed by the initiators during the interaction. This characterisation is represented graphically using a diagram like that shown in Figure 3.4. An example of a protocol definition, corresponding to the *GetInput* protocol of the *Filter* role, is shown in Figure 3.5.

GetInput		
$Filter_i$	$Pipe_i$	
The filter obtains the next data to process		none
		Data

FIGURE 3.5: Example of a protocol definition

3.2.1.3 The model of organisational rules

The model of organisational rules consists of all the organisational rules of the system. Organisational rules are constraints about how the different elements of the organisation interact, and are classified as *liveness* rules and *safety* rules (similarly to the way in which roles' responsibilities are classified, although responsibilities refer to characteristics of only one role, whereas organisational rules refer to characteristics of more than one organisational element). The liveness rules express situations that agents try to bring about, while safety rules state conditions that must be kept invariable. To express organisational rules, Gaia makes use of a language whose details are presented in Section 3.3. The following is an example of a liveness rule:

> *Before being disseminated, any document must be approved by at least three members of the community.*

Similarly, an example of a safety rule is:

> *An agent cannot be, at the same time, the buyer and the seller of a given item.*

3.2.2 The Gaia process

The methodological process of Gaia is divided into three phases: analysis, *architectural* design, and *detailed* design.

3.2.2.1 The analysis phase

The analysis phase deals with collecting the features needed to model the system, and consists mainly of five activities: decomposition of the system into sub-organisations, identification of environmental entities, creation of the preliminary role model, creation of the preliminary interaction model, and creation of the model of organisational rules.

The decomposition of a system into sub-organisations aims to partition the system into more manageable units. Such a decomposition can have as a basis a sub-goal decomposition, the resemblance of a real world structure, the amount of interaction between subsystems, or separation of competences. The phase of identification of environmental entities deals with creating a list of the resources used by the agents while carrying out their activities, but which are not a part of them. Associated with each resource are the rights of access that agents have over it, such as *read* or *change*.

The creation of the preliminary role model consists in the identification of the roles in the system and the construction of their schemata. However, at this early stage it is not necessary (nor appropriate) to produce a complete role model, since the complete definition of the responsibilities can be postponed until the design, when a more detailed view of the system is achieved.

Similarly, the creation of the preliminary interaction model consists in the creation of the definitions of the protocols in the system, placing emphasis on their identification and purpose, more than on their details. In particular, at this stage of the analysis it might be the case that not enough information is available to completely determine the initiators and collaborators.

During the creation of these two preliminary models, it is important to keep the roles and interactions independent of any specific organisational structure. Also, *iteration* at this stage is important: first, the main roles are identified; then, for each role, its associated protocols are documented; this leads to the refinement of the roles model, which in turn can lead to refinements of the interactions model, and so on.

Finally, the analysis phase is completed with the creation of the model of organisational rules, which consists of compiling the rules that govern the behaviour of the system. These rules are based on the roles and interactions identified previously in the preliminary role and interaction models. For example, in an electronic commerce application, an organisational rule might state that an agent cannot play the roles of seller and buyer of the same good at the same time. Organisational rules must restrict the behaviour of agents in order to achieve the overall goals of the system, but at the same time must allow the autonomous, and sometimes self-interested, behaviour of agents. The developer must design the organisational rules of a system in such a way that a good balance between these two conditions is obtained. It must be noted that the organisational rules may be inspired not only by the domain itself, but also by other considerations such as required levels of efficiency or reliability.

3.2.2.2 The architectural design phase

The next phase in the Gaia process, the architectural design, has two main sub-phases, namely the selection of an organisational structure, and the completion of the role and interaction models. As was mentioned previously, during the analysis it is important not to commit to a specific organisational structure. During the design, the selection of the organisational structure of the system is determined. This decision is basically a compromise among different *forces*, each

Characteristic	Meaning
name	name of the service
inputs	information needed
outputs	information produced
pre-conditions	restrictions of use
post-conditions	effects

TABLE 3.2: Characterisation of services in Gaia

pushing in a different direction. These forces are: the complexity of the structure, in terms of computation and coordination; the *distance* from the real-world organisation that the system is modelling; and the need to respect the organisational rules. It is precisely at this stage of the design that organisational patterns are needed, the benefits of which are twofold: they support the decision process, and provide pre-defined organisational structures that can be customised for specific applications.

The second sub-phase in the architectural design, the completion of the role and interaction models, deals with detailing the roles and protocols with the information obtained once the structure is determined. This activity includes the incorporation of new roles and interactions which may have resulted from the application of the previous step. It is suggested that structure-dependent aspects be separated from those independent of the structure to facilitate a possible change of structure.

3.2.2.3 The detailed design phase

The final phase of Gaia, the design phase, consists of producing two models, the agent model and the services model. The agent model indicates which roles will be played by which agents, especially since an agent can comprise one or more roles. The decision of which roles are played by an agent is based on considerations such as efficiency and physical distribution. For example, for efficiency a designer might decide to include three roles in an agent type, instead of having three different agents. The services model is a list of services of all the agents in the system, together with their inputs, outputs, pre-conditions and post-conditions. A service is defined as a single, coherent block of activity in which an agent will engage. The services and their characteristics are derived from the protocols model. Note that, in Gaia, the design is not intended to produce an output detailed enough to be implemented on a particular platform, but focuses on describing how the agents cooperate to achieve the system goals.

3.2.3 Discussion

In conclusion, Gaia is a methodology based on organisational concepts that explicitly considers the three levels of a system organisation: elements (roles and protocols), organisational structure, and the rules that govern its functionality (organisational rules). Also, Gaia presents a clear

separation between the analysis and design phases, as well as between social and internal aspects of agents. Apart from this, Gaia offers the following valuable features.

- It is easy to understand even by non-specialists, since the process is straightforward and the modelling language simple.
- It is also architecture-independent so that no commitment is made to any specific agent architecture, allowing different architectures to be used in the development process.
- Equally important, Gaia is very well known, being one of the most cited (and consequently used) methodologies, and is suitable for extension and enhancement. This is already indicated by the various different extensions that have been built around Gaia itself [131, 77, 50, 16].

However, Gaia is limited in terms of its applicability to the full cycle of development, addressing the analysis and architectural design phases, but leaving the agent design and implementation largely unconsidered. Also, Gaia requires further work; for example, it lacks a catalogue of patterns to support the development of applications and, in particular, it lacks *organisational patterns*, the importance of which is highlighted in the methodology, but no such set is provided. In this chapter we address this omission by describing a framework in which organisational patterns may be developed. However, before proceeding, we first need to address the problem of pattern representation, or finding a suitable form for describing organisational patterns. Since organisational patterns represent organisations, such a description should include their different components, such as organisational rules and structure. Thus, we first need to find appropriate forms for describing organisational rules and organisational structures, which is done in the following sections.

3.3 Organisational rules

3.3.1 Organisations

In the methodological approach described in the previous section, multi-agent systems are modelled by means of organisations, formed of roles and interactions between roles. Roles represent positions or responsibilities within the organisation, whereas interactions are used to accomplish tasks requiring the participation of more than one role. These role interactions give rise to the formation of organisations, each one consisting of a set of organisational rules and an organisational structure, as depicted in Figure 3.6. Organisational rules are constraints on the way agents, roles and interactions relate, and are used to ensure the correct behaviour of the system. Organisational structures, as can be observed in the same figure, consist of topology and control regime, both determined by the role interactions. The former encompasses the communication

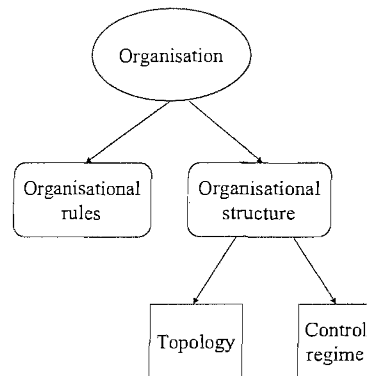

FIGURE 3.6: Components of an organisation

paths between the roles, while the latter comprises the control relationships. Here, the communication path of an interaction is a link between the roles involved in the interaction, whereas the control relationship of an interaction denotes the type of authority — or its absence — between the roles.

If we assume that control relationships are a particular type of constraint between roles, then control relationships can be regarded as organisational rules. Under this perspective, those rules can be seen both as organisational rules or as part of the organisational structure (since control relationships are part of organisational structures). However, in this thesis we have opted for keeping the control regime separate from organisational rules, since the former can be used, on its own, to describe the architecture of a multi-agent system.

In the following, we describe in detail these components of an organisation, and address the problem of characterising and describing them. Specifically, the rest of this section deals with organisational rules, while Section 3.4 deals with organisational structures. The results obtained in these sections are employed in Section 3.5 for the description of organisational patterns.

3.3.2 Overview

As previously seen, organisational rules are a helpful abstraction to make a system exhibit coherent behaviour and achieve its goals, and are present throughout the life cycle of the system. In development, organisational rules are envisaged and documented in the analysis phase, then refined and specified in the design phase, and finally implemented in the implementation phase. Organisational rules also play an important role in determining the characteristics of the system. During development, they are key to determining the organisation of the system, since the *correct* organisation must facilitate the implementation of the organisational rules. At run-time the observance of the organisational rules in part guarantees that the system exhibits the required functionality.

In both stages, development and run-time, it is necessary to express and manipulate the rules. For example, in design the rules are stated unambiguously in the system specification, whereas at run-time the rules are interpreted and evaluated. For this reason, a computational language for expressing organisational rules is required. Although some languages already exist, they are incomplete for our purposes, as we discuss below.

In the literature, two languages have been proposed to specify organisational rules. The first language is presented by Zambonelli et al. in [133] — hereafter called the *Abstract* language — and is based on first-order temporal logic, together with the temporal connectives shown in Table 3.3. The introduction of *time* concepts is necessary since some rules make reference, implicitly or explicitly, to specific times or periods of time. The Abstract language makes use of two predefined predicates, *plays* and *card*, the former of which is applied to an agent and a role and expresses that the agent plays that role, while the latter is applied to a role and denotes the number of agents playing the role at a given moment. Since it is was not the intention of Zambonelli et al. to deal with the details of a formal description of the language, they merely demonstrate its use by means of examples, and so do we. For instance, the formula below makes use of this language to express the organisational rule:

No agent can be both author and reviewer of the same paper

which can also be expressed as:

It is false that at some future moment an agent plays the author role and also plays the reviewer role

or more formally as:

$$\neg\Diamond\ [plays(a, author) \wedge plays(a, reviewer)]$$

The second language for specifying organisational rules appears in [134], in which Zambonelli et al. propose a less formal language, which we will call the *Practical* language, with the purpose of facilitating the specification of organisational rules by non-experienced practitioners. The basis of this language is the use of the operators presented in Table 3.4 for describing how roles can be played by agents. The operator $\rightarrow$ denotes a condition on the order of the roles an agent can play. In general, this operator can be used with the qualifier ^{n} to denote how many times a given role must be played so that another specific role can be played too. The $|$ operator specifies that two roles can be played by an agent at the same time. Finally, the $^{1..n}$ qualifier establishes the number of times a role can be played.

For example, the organisational rule:

No agent can be both author and reviewer of the same paper.

Op.	Meaning	Formula	Satisfied now if ...
$\bigcirc$	next	$\bigcirc\varphi$	φ is satisfied in the next moment
$\Diamond$	sometime	$\Diamond\varphi$	φ is satisfied either now or at some future moment
$\Box$	always	$\Box\varphi$	φ is satisfied now and at all future moments
$\mathcal{U}$	until	$\varphi\mathcal{U}\phi$	ϕ is satisfied at some future moment, and φ is satisfied until then
$\mathcal{W}$	unless	$\varphi\mathcal{W}\phi$	φ keeps satisfied until ϕ is satisfied (which might never happen)
$\mathcal{B}$	before	$\varphi\mathcal{B}\phi$	ϕ is eventually true, and at some time before this, φ is true

TABLE 3.3: Temporal operators

Op.	Meaning	Formula	Satisfied now if ...
$\rightarrow$	played	$R \rightarrow Q$	role Q can be played by an agent only if it played role R before
^{n}	played n-times	$R^n \rightarrow Q$	role Q can be played by an agent only if it played role R at least n times before
$\mid$	concurrency	$R \mid Q$	roles R and Q can be played concurrently
$^{1..n}$	cardinality	$R^{1..n}$	role R must always be played at least once and no more than n times

TABLE 3.4: Practical operators

can be expressed, using this logic, as:

$$\neg(Reviewer(paper(x)) \mid Author(paper(x)))$$

Even though these two languages are useful for some specific situations — such as the specification of organisational rules during design — their applicability is limited, mainly because of lack of completeness. Specifically, these languages do not consider all the elements of a multi-agent system, for example the entities of the environment and other agent features, such as tasks. As a result, these languages leave some relevant relationships between the elements unconsidered, for example the relationship of *utilization* between a protocol and an entity of the environment. Regarding the notion of time, the Abstract language does include temporal operators, but in the Practical language only a particular form of the *before* operator is considered.

Other approaches for organisation modelling have also appeared recently, some of which address aspects closely related to modelling organisational structures, particularly organisational rules. The most relevant approaches, for the purpose of this chapter, are briefly described below.

MOCHA [122] (Model of Organisational CHange using Agents), is a model for specifying organisations, including their changes over time. MOCHA explicitly distinguishes between the *components* of an organisation and the *population* of the organisation. The former characterises

an organisation as a set of *actions* and a set of *roles*. Here, actions have *duration* (in arbitrary units) and can be decomposed into *sub-actions*, while roles maintain relationships of *obligation* (a role is obliged to execute an action specified by another role, within the corresponding duration) and *influence* (a role influences another role to adopt an obligation) between them. The population of an organisation, which is not considered as part of the organisational structure, specifies which roles are played by which agents, as well as the *attitude* of each agent to enact the obligations and influences related to the corresponding roles. Regarding the specification of how the different elements of an organisation are allowed to relate (organisational rules), MOCHA provides a simple mechanism based on the duration of actions, and constraints on the times that actions start and end. By carefully setting the values of such durations and constraints, a desired sequence of actions can be established and potential conflicts can be avoided. As can be observed, this approach concerns mainly the order of actions, but leaves unconsidered other types of organisational rules, for example cardinality of roles.

In addition, several approaches for organisational modelling have been presented in the workshop series on Coordination, Organisation, Institutions and Norms in agent systems (COIN). In [58] Grossi et al. define an organisational structure as a 4-tuple consisting of a finite set of roles and three binary relations on roles: *power*, *coordination* and *control*. Power denotes that an agent (enacting one role) can delegate a goal to another agent (enacting another role). Coordination refers to the issue of the information needed by one agent (enacting a role) for achieving its goals and complying with the norms of the organisation. Finally, control involves monitoring activities to determine failures to achieve goals or violations to norms. As can be noted from this brief review, this definition does not consider explicitly organisational rules to constrain the way in which different elements of the organisation are allowed to relate.

Montealegre et al. [123] present a model for hierarchical organisations based on norms. This model, expressed in UML, extends a normative framework for agent-based systems [87] with organisational elements, some of which are specific to hierarchical structures. In this model, norms are used to regulate the behaviour of the members of an organisation (like organisational rules are used in this thesis) and consist of seven elements: the goals prescribed by the norm, the agents that must comply with these goals, the agents that might benefited from compliance with the norm, the conditions to activate the norm, the conditions under which an agent is not obliged to comply with the norm, the punishments for not satisfying the goals, and the rewards for complying with the norms. Goals, which form the core of norms, are described by means of functions (for example *read(paper)*), but no details are provided about their representation.

In the same field of normative organisations, *deontic* logics (that is, logics concerned with obligation, permission and related concepts) have been used to express norms that regulate the behaviour of the members of an organisation [86, 121].

3.3.3 Requirements of a language for organisational rules

This identification of drawbacks in existing specification languages naturally leads us to establish the characteristics that such a language must meet. To begin with, since organisational rules are constraints on the elements of a multi-agent system, they encompass a huge number of situations. For example, some of the situations encompassed by rules referring *only* to roles include: the number of agents that play it at the same instant or along the history of the system; the relative order in which the role is played with respect to other roles of the same agent or other agents; mutual exclusion with other roles of the same agent or other agents; the number of times the role is played for a given agent or for the whole system; and so on. Since it is difficult to foresee all the possible situations encompassed by organisational rules, the language must be general enough to cover the most common situations, and must allow extensions to cover other specific situations. We will call a language with this property *expressive*.

In principle, organisational rules can be described using simple natural language text. However, such an ambiguous description is inappropriate for tasks such as the design, specification and implementation of systems, or for manipulating organisational rules, as is the case when organisational rules are interpreted and evaluated at run-time. In order to accomplish these tasks a *manipulable* language is needed.

Apart from being expressive and manipulable, such a language must satisfy the following requirements.

1. The language must consider all the elements of a multi-agent system that are relevant for constraining the behaviour of the agents. This is required because organisational rules can make reference to any element in the system.

2. Similarly, the language must consider all the meaningful relations between these elements; for example, the relation *plays* between an agent and a role (when an agent plays a role), and the relation *initiates* between an agent and a protocol. Some of these relations can be an integral part of the language, and some can be added to the language for specific applications.

3. The language must consider the representation of time, since many types of organisational rules relate events that occur at different points of time, as in the following rule:

   To play role r an agent must have played role q before

4. The language must include the elements commonly found in predicate logic, such as the logical connectors *and*, *or* and *not*, since organisational rules are predicates that can be evaluated to decide whether they are *true* or *false*. Also, the language must include the existential quantifier, as a means to refer to an unspecified element, as well as a universal quantifier, to refer to multiple elements at the same time, as is the case for the environmental entities *papers* in the following rule:

Every *paper must be reviewed by three different reviewers*

5. Finally, the language must be intuitive and easy to understand, since it is intended to be used by an *average* software developer.

3.3.4 A language for organisational rules

To overcome the limitations of current approaches, we envisage a language based on the Abstract language mentioned above, together with explicit inclusion of the entities of the environment, as well as a characterisation of the relationships between the different types of elements. The Abstract language was selected over the Practical language because it can be more easily extended to consider other elements, and the notion of time is more explicit. The resulting language, which we call LEVOR (language for the expression and verification of organisational rules) is detailed below.

Basically, LEVOR is a language based on first order predicate logic that uses temporal operators. Its complete syntax is presented in Table 3.5, in which the entry symbol has been represented by $< OrganisationalRule >$, the non-terminal symbols are surrounded by angular brackets, the terminal symbols are in quotes, and the null string has been denoted by λ.

LEVOR allows the expression of organisational rules as propositions that can be evaluated at any moment to determine its logical value: true or false. In the former case the rule is said to be *observed*, while in the latter it is said to be *violated.* Such propositions can be formed by using the classical logical operators (not, and, or, implies), and the existential and universal quantifiers, all with their usual meaning. Also, the proposition can contain the temporal operators mentioned in Section 3.3.2, whose meaning and syntax appear in Table 3.3. As can be observed, *next*, *sometime* and *always* are unary operators, while *until* ($\mathcal{U}$), *unless* ($\mathcal{W}$) and *before* ($\mathcal{B}$) are binary operators. In the LEVOR language, time is considered to be a sequence of discrete points.

The elements of the language consist of integers, the logical constants *true* and *false*, other constants, variables, functions, and predicates. These other constants refer to names of roles, agents, protocols, resources, parts of resources, and activities, and are denoted by plain strings, with the exception of protocols and parts of resources. Protocols are denoted by their *name*, *initiator*, *collaborator* and the *data* required or produced, as in

$$SubmitPaper(author, collector, paper).$$

The different parts of a resource are referred to by using a *dot notation*, for example the element *author* of the resource *paper* is denoted by *author.paper*. Activities in LEVOR are also denoted by means of the dot notation; for example, *filter.keepFlow(f)* denotes the activity *keepFlow* of role *filter*, whose parameter is *f*.

< *OrganisationalRule* >	::=	< *Proposition* >
< *Proposition* >	::=	< *AtomicFormula* >
	|	< *UnaryLogicalOp* >< *Proposition* >
	|	< *Proposition* >< *BinaryLogicalOp* >< *Proposition* >
	|	< *Quantifier* >< *Variable* > "(" < *Proposition* > ")"
	|	< *UnaryTemporalOp* >< *Proposition* >
	|	< *Proposition* >< *BinaryTemporalOp* >< *Proposition* >
< *AtomicFormula* >	::=	< *Term* >< *ComparisonOp* >< *Term* >
	|	< *PredicateSymbol* > "("
		< *Term* > ("," < *Term* >)* ")"
	|	True
	|	False
< *Term* >	::=	< *Variable* >
	|	< *FunctionalTerm* >
	|	< *Constant* >
< *FunctionalTerm* >	::=	< *FunctionSymbol* > "("
		< *Term* > ("," < *Term* >)* ")"
< *Variable* >	::=	< *AgentSymbol* >
	|	< *RoleSymbol* >
	|	< *ResourceSymbol* >
< *Constant* >	::=	Integer
	|	< *ProtocolConstant* >
	|	< *PartConstant* >
	|	< *ActivityConstant* >
	|	< *AgentSymbol* >
	|	< *RoleSymbol* >
	|	< *ResourceSymbol* >
< *ProtocolConstant* >	::=	< *ProtocolSymbol* > "("
		< *RoleSymbol* > "," < *RoleSymbol* >
		("," < *ResourceSymbol* >)* ")"
< *PartConstant* >	::=	< *ResourceSymbol* > "." < *ResourceSymbol* >
< *ActivityConstant* >	::=	< *RoleSymbol* > "." < *ActivitySymbol* > "("
		< *Resources* > ")"
< *Resources* >	::=	< *ResourceList* >
	|	λ
< *ResourceList* >	::=	< *ResourceSymbol* >
	|	< *ResourceSymbol* > "," < *ResourceList* >
< *UnaryLogicalOp* >	::=	"$\neg$"
< *BinaryLogicalOp* >	::=	"$\wedge$" | "$\vee$" | "$\Rightarrow$"
< *Quantifier* >	::=	"$\exists$" | "$\forall$"
< *UnaryTemporalOp* >	::=	"$\bigcirc$" | "$\Diamond$" | "$\Box$"
< *BinaryTemporalOp* >	::=	"$\mathcal{U}$" | "$\mathcal{W}$" | "$\mathcal{B}$"
< *ComparisonOp* >	::=	"=" | "$\neq$" | ">" | "<" | "$\geq$" | "$\leq$"
< *PredicateSymbol* >		
< *FunctionSymbol* >		
< *ProtocolSymbol* >		
< *AgentSymbol* >		
< *RoleSymbol* >		
< *ResourceSymbol* >		
< *ActivitySymbol* >		

TABLE 3.5: Grammar for the LEVOR language

Function	Target	Syntax
card	Role	card(role_name)
card	Protocol	card(protocol_name)

TABLE 3.6: Pre-defined functions in the LEVOR language

Variables in LEVOR are used to denote agents, roles and resources. Although both variables and constants are represented by strings, they are differentiated by how they are quantified: elements referred to in the quantifiers are variables, whereas all the other elements are considered constants. For example, in the organisational rule below, a is a variable, while $Agent$, $SeniorBuyer$ and $JuniorBuyer$ are constants.

$$\forall a : Agent \ \ (plays(a, JuniorBuyer) \ \mathcal{B} \ plays(a, SeniorBuyer))$$

Functions are used to represent properties of elements. For example, the cardinality of a role r, denoted as *card(r)*, is the number of agents playing the role. Similarly, the cardinality of a protocol p is the number of times that the protocol has been initiated and is denoted by *card(p)*. Table 3.6 summarises these functions of the language.

Predicates are used to express relations between different organisational elements, evaluating to *True* if the relation holds, or false otherwise. *plays* is the most basic relationship involving agents and roles. We can express that $agent1$ plays role r by writing *plays(agent1, r)*.

Regarding roles and protocols, *initiates(r, p)* denotes that role (or agent) r commences the execution of protocol p, and *participates(r, p)* denotes that role r initiates or collaborates in the execution of protocol p. If the initiator itself is not relevant, the relation *initiated(p)* can be more convenient. In addition, *terminated(p)* indicates the end of execution of protocol p.

Relationships between roles and environment are represented through these generic relations: *reads(role, data)* and *modifies(role,data,value)*, which represent that a role reads (senses) and modifies (acts on) a resource, respectively. Table 3.7 summarises the predicates of the language.

Predicate	Argument 1	Argument 2	Syntax
plays	Agent	Role	plays(agent_name, role_name)
initiates	Role	Protocol	initiates(role_name, protocol_name)
participates	Role	Protocol	participates(role_name, protocol_name)
initiated	Protocol		initiated(protocol_name)
terminated	Protocol		terminated(protocol_name)
reads	Role	Resource	reads(role_name, resource_name)
modifies	Role	Resource	modifies(role_name, resource_name, new_value)
terminated	Activity		terminated(activity_name)

TABLE 3.7: Pre-defined predicates in the LEVOR language

The following are some examples of organisational rules and how the language is used to express them.

1. An agent cannot be a senior buyer if it has not been a junior buyer before.

$$\forall a : \mathit{Agent}\,(plays(a, JuniorBuyer)\ \mathcal{B}\ plays(a, SeniorBuyer))$$

2. There must be at least 10 different reviewers of papers.

$$card(Reviewer) \geq 10$$

3. An agent cannot be a buyer and a seller at the same time.

$$\forall i : Item(\forall x : Buyer(\forall z : Buyer(\forall y : Seller(\forall w : Seller$$

$$(\ initiates(a, Buy(x, y, i)) \wedge$$

$$initiates(b, Sell(w, z, i)) \Rightarrow a \neq b)))))$$

4. The selection process begins when all the papers have been reviewed three times.

$$\forall p : Paper(\forall r : Reviewer(\forall c : Collector(\ card(ReviewPaper(r, c, p)) = 3$$

$$\mathcal{B}\ initiated(SelectPapers))))$$

5. Every paper must be reviewed exactly three times.

$$\forall p : Paper(\forall r\ Reviewer(\forall c\ Collector((card(ReviewPaper(r, c, p)) = 3))))$$

In summary, we have presented a language to specify organisational rules. We have clearly established the terms and operators of the language, and have provided examples that show its use. This language can be used to specify the organisational rules in the analysis and design of a multi-agent system. Also, since the language is manipulable, it can be used in run-time tasks such as monitoring the observance of organisational rules. The advantages of our proposal over existing proposals are that we have limited the scope of the language, have explicitly considered all the organisational elements of a multi-agent system and have characterised the relationships between the elements of the system. The resulting language covers many practical situations, but can easily be extended for other situations not considered here.

3.4 Organisational structures

3.4.1 Introduction

Organisational structures are often described in the literature either by means of figures or textual description [134, 64]. Such a rough form of description relies on our intuition and our knowledge of their real-world counterparts. Thus, for example, when the term *hierarchical organisation* is used, most of us guess its meaning by association to the structures commonly found in businesses. Although this form of description is adequate for communicating ideas between humans, for other purposes a more exact and complete characterisation is needed, as is the case when attempting to select the most appropriate structure for a multi-agent system, or when checking the observance of the structure at run-time. In this section we tackle this problem by first stating the features that such a characterisation must possess, then reviewing current characterisations, and finally providing a model for organisational structures from which a characterisation is obtained, as well as a language for their specification.

As indicated above, when organisational structures are represented by means of figures or, more specifically, graphs, roles are depicted as nodes and their interaction as arcs joining them. Although this provides a good indication of the overall structure, it has the following drawbacks. First, the figures can become burdensome in the case of big organisations. Second, in these figures the control regime of the organisation is often omitted because labelling each arc with its corresponding control relationships makes the graph illegible. Finally, and most important, figures are useless for tasks in which some form of computational manipulation of the structure is needed; for example, checking that the structure is not violated at run-time or creating a catalogue of structural designs. These limitations can be overcome by using a *non-graphical* model to describe the structure. In particular, we want to be able to: provide a definition for the term *control regime*; describe organisational structures in an unambiguous way, including their topology and control regimes; perform basic manipulation on the structures such as their storage and retrieval in memory and files; and check at run-time if the structure of the organisation is being respected. For this to happen, such a representation must exhibit the following features.

- Since an organisational structure comprises topology and control regime, the representation must be expressive enough to specify the topology of the structure as well as the authority relationships between the roles.
- The representation should be simple and easy to use by practitioners because we aim to provide development tools for non-agent specialists.
- The representation must allow the identification not only of primitive elements of the structure, which are roles and authority relations, but also of subsets of primitive elements. For example, we need to represent the roles that satisfy certain conditions, as in *the apex has a relationship of authority over* all *the other roles of the structure.*

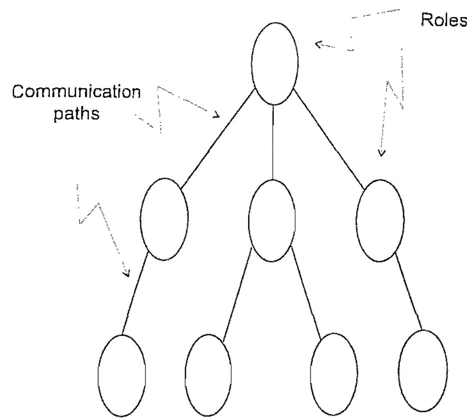

FIGURE 3.7: Topology representation

- Since the topology of an organisation resembles a graph formed of nodes and arcs, the representation should allow the expression of properties of an organisational structure by means of referencing nodes and arcs, and *vice versa*. In other words, the representation must allow to mix analytical and graph-based concepts, as in the expression:

  each controller maintains a peer to peer relationship with the controllers with which it communicates.

3.4.2 Characterisation and informal analysis

Organisational structures encompass two aspects: *topology* and *control regime* ([45]). The topology of an organisation is formed of all the communication paths between the member roles. Conversely, the control regime refers to the control relationships between the member roles. Control relationships denote the form in which one role influences the behaviour of the other. Common control relationships are *peer-to-peer*, in which no role is subordinated to another, and *master-slave*, where the existence of one role is justified only in terms of supporting another role.

3.4.3 A model for organisational structures

Below, we propose a non-graphical representation of organisational structures. First, we provide a characterisation of organisational structures. Then, we outline the language in which the characterisation can be expressed. Finally, we include an example of the representation of a simple structure.

3.4.3.1 Characterisation of organisational structures

The characterisation of organisational structures contains the description of its elements and the way in which they are related. We define an organisational structure as a 5-tuple of the form $(\mathcal{P}, \mathcal{C}, \mathcal{R}, \mathcal{L}, \mathcal{A})$, whose entries are the sets described below (a summary of the notation used can be found in Table 3.8). The first three entries refer to the *analytical* description of the structure, while the last two entries link the analytical characterisation to a graph of nodes and arcs.

1. $\mathcal{P}$, the *participants set*, consists of the list of roles in the organisation.

2. $\mathcal{C}$, the *control relationships set*, lists the control relationships used in the structure. The members of this set can be defined as needed for a specific application or can be taken from a general set of *predefined relationships* (for a list of predefined relationships see Table 3.9).

3. $\mathcal{R}$, the *control regime model*, is a relation between two members of the participants set and a member of the control relationships set $\mathcal{R} \subseteq \mathcal{P} \times \mathcal{P} \times \mathcal{C}$. An element of the relation means that the first role has the specified control relationship with the second role. $\mathcal{R} = \{(r, s, c) \mid r, s \in \mathcal{P} \wedge c \in \mathcal{C} \wedge$ there is a control relationship of type c between r and $s\}$

4. $\mathcal{L}$, the *labels set*, is a list of strings that name the nodes of the graph in a unique form; for example:
$$\{head, left_leaf, right_leaf\}$$
or
$$\{stage(i) \mid i = 1..3\}$$
Graphically, each label is put near and outside the corresponding node.

5. $\mathcal{A}$, the *association model*, is a relation from the participants set into the labels set ($\mathcal{A} \subseteq \mathcal{P} \times \mathcal{L}$). Thus, its members are ordered pairs consisting of a role and a label, which means that the role is associated to the node identified by the label: $\mathcal{A} = \{(r, l) \mid r \in \mathcal{P} \wedge l \in \mathcal{L}\}$. Examples of association models include the following:
$$\{(manager, root), (buyer, left_leaf), (seller, right_leaf)\},$$
which means that the roles manager, buyer, and seller occupy the nodes $root$, $left_leaf$ and $right_leaf$, respectively; and
$$\{(controller(i), stage(i)) \mid i = 1..3\}$$
meaning that each controller occupies its corresponding stage node in the graph. Graphically, the name of the role is placed inside the corresponding node.

Entry	Denotes
$\mathcal{P}$	Participants set
$\mathcal{C}$	Control relationships set
$\mathcal{R}$	Control regime model
$\mathcal{L}$	Labels set
$\mathcal{A}$	Association model

TABLE 3.8: Summary of notation

Relationship	Description
authority	permanent authority of one role over the other
peer	no permanent authority
dependency	dependence on information provided by the other role

TABLE 3.9: Pre-defined control relationships

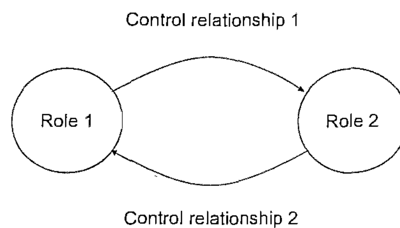

FIGURE 3.8: Nodes linked by more than one arc

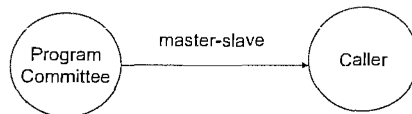

FIGURE 3.9: Using arrows to denote control relationships

Additionally, an element of the control regime $\mathcal{R}$ (whose meaning is that the first role has the specified control relationship with the second role) can be represented by a labelled directed arc from one role to another in which the label corresponds to the control relationship.

Note that any two roles can be linked by zero, one or two relationships. Accordingly, the corresponding nodes can be linked by one or two arcs (as in Figure 3.8, in which the nodes are linked by two arcs) or not linked at all. On the other hand, nodes are commonly occupied by only one role since there is no clear benefit in having more than one role in a node.

In general, the control relationships are not symmetric, so the order of the roles is important; for example, $master_slave(a, b)$ is different from $master_slave(b, a)$. For this reason, arcs with arrows are used in the graph, as in Figure 3.9, where the arrow indicates that the Program-Committee is the master and the Caller is the slave. If the control relationship is symmetric, the arrows of the arc may be omitted.

3.4.3.2 Characterisation of control relationships

Although the number of different control relationships is unlimited, a small subset of control relationships may be enough to represent the most common types of interaction. In the following we present just such a small subset of these relationships. We choose these relationships because they are simple to understand and together cover most situations.

- $authority(r, s)$ holds if role r has some type of control over role s. The control may be a expressed as a *master-slave* relationship, *task allocation* (such as *work partition* or *work specialisation*), *order*, or any other relationship denoting authority (that one role agrees to execute the orders given by the other).
- $peer(r, s)$ holds if no role has authority over the other, if the authority position is alternated, or if their authority is based on *requests* (with no obligation to commit) or on a *trust* relationship.
- $dependency(r, s)$ holds if role s needs information provided by role r to accomplish its responsibilities.

These relationships are summarised in Table 3.9. They are neither exclusive nor exhaustive, and this *default* set of relationships can be replaced with a more appropriate one for each specific application. Regarding the problem of how to identify the control relationship that best describes the interaction between two roles, we envisage two approaches based on the strong ties between protocols and control relationships (as described below). First, if the protocols between the nodes have been defined, they can be used to determine the control relationship. For example, if the messages exchanged between two nodes refer to requesting and providing information, their control relationship can be described as *dependency*. The details of this approach depend on the specific implementation. For instance, assuming a FIPA-compliant implementation, the use of the *FIPA-query* protocol would suggest a *dependency* relationship, since the main objective of the protocol is to obtain information, presumably to complete a task. On the other hand, the use of the *FIPA-request* protocol would suggest an *authority* relationship in which one role *requests* the other role to carry out an activity.

Second, a different approach is used when the control relationships are defined before the protocols, for example when reusing a previous organisational structure. In this case, the control relationships guide the design of the protocols.

Whichever the approach, protocols and control relationships must be consistent, and not in contradiction. For example, if the control relationship between two roles has been defined as *authority*, then those protocols in which the subordinated role *orders* the main role to perform an action are not allowed. A procedure to check such consistencies at design time can be based on an information repository containing the control relationships, the type of messages allowed for each of them, and in some cases the content of the messages. Although such a procedure is straightforward in itself, determining the content of the repository requires careful inspection of the meaning of the messages and their relation to the control relationships. Although a detailed presentation of such a procedure is not included here because it depends on the particular set of messages employed, a simple example is presented below.

Consider an application in which FIPA-type communicative acts [35] are used to express the messages between the roles. In an *authority* relationship between two given roles, some commanding communicative acts might not be used unrestrictedly in messages from the subordinated role to the other role, since this would violate the nature of the authority relationship. For instance, the *refuse* act (which denotes the action of refusing to perform a given action, and explaining the reason for the refusal) would be allowed only for valid reasons, such as unavailability of a service. Other communicative acts that might be totally or partially disallowed are *request* (request to perform some action), *propagate* (send the received propagate message to other agents), *reject proposal* (rejecting a proposal to perform some action during a negotiation), and *subscribe* (requesting a persistent intention to notify whenever a selected object changes).

3.4.3.3 Language for expressing organisational structures

It can be observed that in an organisational structure $(\mathcal{P}, \mathcal{C}, \mathcal{R}, \mathcal{L}, \mathcal{A})$, the sets $\mathcal{L}$ and $\mathcal{A}$ are significant only in a graphical sense, but the core of the structure is contained in the $\mathcal{P}$, $\mathcal{C}$ and $\mathcal{R}$ sets. Furthermore, the elements of $\mathcal{P}$ and $\mathcal{C}$, (participants and control relationships, respectively) can be obtained from the elements of $\mathcal{R}$, the control regime model. This is why we define the control regime model as the *language* to express an organisational structure. In other words, an organisational structure is defined by its control regime model. Just like any set, the control regime model is expressed by listing its elements (in the form $controlrelationship(role1, role2)$); for example,

$$\{authority(manager, buyer), authority(manager, seller)\}\ ,$$

or, equivalently, by providing a description of their elements, as in:

$$\{peer(controller(i), controller(i+1))|i = 1..3\}\ .$$

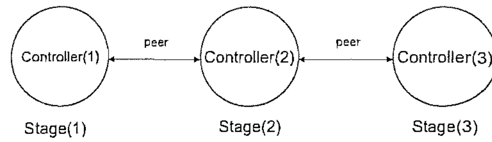

FIGURE 3.10: Graphical representation of a pipeline structure

Model	Instance
Organisational structure	$\{peer(Controller(i), Controller(i+1)) \mid i = 1..2\}$
Participants set	$\{Controller(1), Controller(2), Controller(3)\}$
Control relationships set	$\{peer\}$
Labels set	$\{\text{“}Stage(1)\text{”}, \text{“}Stage(2)\text{”}, \text{“}Stage(3)\text{”}\}$
Association model	$\{(Controller(i), \text{“}Stage(i)\text{”}) \mid i = 1..3\}$

TABLE 3.10: Formal definition of a pipeline

3.4.3.4 Example

In this section a *pipeline* structure is used to exemplify the model for organisational structures presented above. In the first part, the pipeline is intuitively introduced, in the second part a graphical representation is shown and, finally, in the third part, its definition and characterisation are presented.

1. A pipeline structure resembles a manufacturing pipeline for producing items or goods. Every raw item enters the pipeline and, at the end, the item, fully processed, is obtained. Such processing is divided into several independent stages, arranged in sequence, with each stage enhancing the item in a particular way. We model the functionality at stage i by means of a role, namely $controller_i$. In addition, the independence of the stages is modelled by a control relationship of type *peer*.

2. The graphical representation of such a pipeline (of three stages) is depicted in Figure 3.10, which shows the three nodes in circles with their labels below, the roles names inside the circles, and the two control relationships linking the nodes.

3. The expression that defines the organisational structure of this pipeline of three elements is presented in Table 3.10, together with its corresponding set of participants, and set of control relationships. The table also presents a possible labels set, and the corresponding association model, to link the the structure with the graphical representation.

3.4.3.5 Conclusions

In summary, we have presented a model for the characterisation of organisational structures that includes a language to describe them. This form of representing structures is helpful for automatising tasks such as storing and retrieving structures, deciding if two structures are *equivalent*, and monitoring the observance of the structure at run time. Also, a graphical counterpart of a structure can be easily obtained from this representation, which is valuable for use by mainstream software developers, for whom visualisation is an important tool.

3.5 Organisational Patterns

3.5.1 Introduction

The use of *organisational patterns* [56] is highlighted in the methodological process of Section 3.2 as a key part in the design of a multi-agent system. There, it is observed that before selecting a pattern the developer has already identified the main roles and interactions needed to accomplish the system goals, and has also identified the organisational rules. At this stage, the developer uses a library of patterns to decide the *best* structure for the system. The library provides the following benefits to the developer, all oriented towards facilitating the development process.

- Patterns help to reduce ambiguity. While the developer may sketch the organisational structure using just an informal diagram, patterns are specified according to a more structured and unambiguous description.
- Patterns help to isolate the application-independent features of an organisation from those applicable only in specific situations. This promotes re-usability and increases development speed.
- Patterns help to avoid platform or technology dependence at this stage of the design process. Nevertheless, they provide general guidelines on matters related to implementation.
- Patterns include a description of the situations in which their use is recommended, so that matching a specific application with one of these situations, a pattern appropriate for that application can be selected.
- Patterns present a more detailed description of the structure than one that developers would normally achieve at this stage of the design; for example, the list of organisational rules corresponding to the management of the organisation is not normally included at this point, but should appear in a pattern.

After selecting the right pattern, the developer must proceed to complete the role and interaction models by adapting the pattern to the particular situation, resulting in the final role and interaction models. This may involve the creation of new roles and interactions, as well as detailing those identified previously.

It should be noted that the main question when selecting one of these patterns is what organisational structure best models the characteristics of the system-to-be. As pointed out in [133], such a structure must not only appropriately describe the characteristics of the system but must also take into account issues like efficiency and flexibility. According to Fox [45], when designing a distributed system, one must consider two issues: task decomposition and selection of a control regime. In Gaia, a preliminary task decomposition is undertaken in the analysis phase, but the decision of the definitive *topology* is postponed until design. Thus, the selected pattern must provide the topology and the control regime for the organisation.

A first attempt to create a set of patterns may be to take all possible combinations of known topologies and control regimes. This, of course, would lead to an unmanageable number of patterns. Another approach is to consider only those combinations that are potentially useful, either based on experience, or by analogy to other areas in which organisational structures have been applied. In our work, we assume that a small number of such organisational patterns would suit a broad range of applications. In addition, each pattern in the set represents a *family* of organisational structures, rather than a specific instance.

3.5.2 Pattern layout

The form in which patterns are described, which we will refer to as their *layout*, is almost as important as patterns themselves. A good form of description makes patterns easy to understand and use. On the other hand, a bad form of description defeats the main purpose of patterns, which is to *facilitate* the development process; it may lead to misinterpretation, a waste of time, and eventually discourage the use of patterns.

3.5.2.1 Pattern requirements

Since no layout for organisational patterns exists to date, we must develop one suitable for our purposes. Before presenting our layout for organisational patterns, we consider the requirements of such a layout. First, basically a pattern is a *solution*, so a pattern layout is a form of expressing a solution. In our context, such a solution expresses how a particular organisational structure can be used to model the operation of a multi-agent system. To model the operation of this system, two aspects must be covered, the static structure and the dynamics, or the way the components interact at run-time to achieve *meaningful* behaviour. Second, since a self-contained layout is desirable, in addition to describing a solution a layout must include contextual information such as the problem being solved, exemplar situations, advantages and disadvantages of its use, and

so on. Nevertheless, several of these contextual sections can be identical for some — or all — of the patterns. Third, specifically for our purposes, the layout must facilitate the use of the patterns as part of the methodological process. Finally, the layout must include a *unique* name for identification purposes, and a description summarising the main characteristics of the pattern.

Pattern descriptions are divided into *sections*. We determine the sections and their content in our layout based on two notions. The first is the Context-Problem-Solution metaphor cited in [11], and the second is the form in which design patterns [48] — particularly agent patterns [26, 84] — are described. The former, the Context-Problem-Solution metaphor, states that the essence of a pattern relies on the relationship between the problem, the situations in which it commonly occurs, and its solution. In other words, every pattern description must include these three aspects arranged in such a way that it clearly shows the problem in question, the context in which the problem exists and the solution provided.

Regarding the second notion on which we base our layout, the way other agent patterns are being described, we observe that in [26] the following sections are suggested as mandatory in any layout: *name*, *context*, *problem*, *forces* and *solution*. Apart from these, *rationale* and *known uses* are also included specifically for the description of coordination patterns. Similarly, the layout employed in [84] consists of one part common to all the patterns and another part specific to each of the categories considered. The common part includes: *name*, *alias*, *problem*, *forces*, *entities*, *dynamics*, *dependencies*, *example*, *implementation*, *known uses*, *consequences*, and *see also* (the meaning of these sections is given below). We can observe, in these two cases, that the layout of agent patterns is divided into two parts: a general part, that deals with the identification of the patterns, the statement of the problem they solve, the context in which they are used, and general aspects of the solution they offer; and a specific part, that deals with those aspects of the solution that are applicable only to that particular type of patterns.

However, there is no common agreement about what constitutes a good pattern description. For instance, using a unique layout to describe patterns of different types has both disadvantages and advantages. On the one hand, doing this could result in a superficial description that ignores essential details that make a pattern distinct. On the other hand, different descriptions would make it difficult to compare patterns when selecting one for a specific application. In the agent-oriented approach it is even less clear what a good pattern description is, mainly because of the lack of consensus over agent terms and the diversity of agent-oriented methodologies.

3.5.2.2 An organisational pattern layout

Thus, based on the Context-Problem-Solution metaphor and on how similar agent patterns are described, our layout is divided into two parts. It includes a general part, similar to those found in other pattern descriptions, and a particular part, which is specific to organisational patterns. As discussed above, this particular part is formed of specific issues concerning the solution of

Section	Description
*N*ame	a unique identifier for the pattern
*A*lias	other names to denote the pattern
*P*roblem	the problem to which the pattern provides a solution
*C*ontext	the situation that surrounds the problem
*F*orces	factors that determine whether to apply the pattern
*S*olution	a textual description of the solution provided
*R*estrictions	conditions for the pattern to be applicable
*C*onsequences	advantages and disadvantages of using the pattern
*I*mplementation	brief guidelines toward implementing the pattern
*B*ased on	traditional patterns on which the pattern is based
*R*oles	the roles in the organisation
*E*nvironmental entities	the resources in the system employed by the roles
*S*tructure	the structure of the organisation
*R*ules	the organisational rules of the organisation
*D*ynamics	the way the organisation operate at run-time

TABLE 3.11: Summary of the layout for describing organisational patterns

the problem, namely the static structure and the dynamics of an organisation and, consequently, it consists of these sections: *roles*, *structure*, *rules* and *dynamics*.

The sections of the pattern layout are presented below, and a summary of the layout is given in Table 3.11.

- Name: short descriptive name for the pattern.
- Alias: other names by which the pattern may be known.
- Context: a description of the situation in which the pattern is applicable. Note that the context is a general description, and alone is not sufficient to determine the applicability of the pattern. To this end, the context is complemented with *forces* (see below).
- Problem: the problem solved by the pattern. It basically takes the form of a search for an appropriate organisational structure to model an agent-based system.
- Forces: description of factors that influence the decision as to when to apply the pattern in a context. Forces push or pull the problem towards different solutions or indicate possible trade-offs [26]. We identify the following forces in organisational patterns.
  - Coordination efficiency: the structure of an organisation strongly influences its efficiency for coordination tasks in terms of information shared and number of messages interchanged.
  - Coupling: the degree of interdependence between the roles. Although coupling is inherent in all structures, it varies in degree. A structure with high coupling is difficult to extend.

  - Subordination relationships: some structures impose specific control regimes on their roles, which may not be appropriate for some situations.
  - Topology complexity: simple topologies exhibit low coordination overhead but require powerful roles in terms of resources and task processing.

- Solution: a textual description of the solution.
- Restrictions: scope of the pattern.
- Consequences: side-effects of using the pattern, including advantages and disadvantages.
- Implementation: brief advice on how to implement the pattern.
- Roles: the participating roles and their characteristics. The roles in the pattern are described by means of *role schemata* (see Section 3.2).
- Structure: the topology and the control regime of the organisation. We use the model presented in Section 3.4.3 to describe this section of the pattern.
- Environmental entities: the resources or information that the roles use while carrying out their tasks, but are not an integral part of them. The roles interact with the environmental entities through sensing (reading) and affecting (modifying) them.
- Dynamics: the dynamics encompasses the way in which the roles interact to solve the problem. The interactions between the roles are described using protocol definitions (see Section 3.2) and AUML-style sequence diagrams (see Section 2.5.3.1). It is usually the case that the dynamics can be decomposed into *scenarios*, each representing a meaningful behaviour with distinct results. For example, one scenario can deal with the *normal*, or expected, form of operation, while some others may be related to singular situations, for instance exception management.
- Rules: constraints to be respected in the organisation independent of the application domain. The language used to express the rules is presented in Section 3.3.4.

This layout is used in the following to describe three instances of organisational patterns. Note that, since these patterns solve essentially the same problem and have the same context, the content of some sections is the same for all.

3.6 Catalogue of patterns

3.6.1 Introduction

This section presents a catalogue of organisational patterns that consist of three representative cases, covering a range of different situations. The *pipeline* is simple in concept and structure, while the *hierarchy* is flexible and resembles real organisations, and the *marketplace* exemplifies open organisations.

3.6.2 The pipeline pattern

According to the number of different roles and communication paths between them, the *pipeline* is one of the simplest types of organisation. A pipeline is a structure that processes (produces, transforms or augments) items in a series of steps, or stages. The stages are arranged in sequence, so that the process of a given item initiates at the first stage, continues at the intermediate stages and is completed at the last stage. For any item, each stage depends on the finalisation of the process of the previous stage to carry out its own process. However, two or more stages can process (different items) at the same time. In this pattern, we call the entities that perform the process in each stage *filters*, and the maximum rate at which a filter can process items *flow rate*. The entities that link two stages are called *pipes*, and although their main function is to serve as a communication link between the stages, they can also perform more complex tasks, such as compensate for any difference in the flow rate of the filters, and notify the filter that new data is available. In a pipeline, the coordination consists of ensuring that the flow rates of all the stages are similar, so that no bottlenecks occur. In addition, to alleviate the occurrence of bottlenecks, buffers can be used.

The organisational pattern corresponding to a pipeline structure is as follows.

- Name: Pipeline.
- Alias: Flat.
- Context. According to the Gaia process, before selecting a pattern the developer has already completed the roles and the interactions models, and has also compiled the organisational rules and defined the organisational structure (topology and control regime). After selecting the appropriate pattern, the developer must be ready to complete the final roles and interactions models.
- Problem. The problem addressed by this pattern is finding an organisational structure that best describes the system under development. The analysis of the system has already produced the preliminary role and interaction models, and what is missing is to define the topology and the control regime of the organisational structure. In addition, the following characteristics of the system have been identified. First, the problem consists of (or can be modelled as) a manufacturing process in which the overall goals are achieved by a strong collaboration among the participating roles. Second, such a collaboration can be seen as a processing line in which each role performs a transformation on some given information and delivers it to the next member of the line.
- Forces:
  - Coordination efficiency: low.
  - Coupling: low.
  - Subordination relationships: none.

- Topology complexity: very simple.

- Solution. The pipeline has been used extensively in mainstream software engineering to design applications in which the overall processing can be decomposed into independent sequential tasks. The tasks are performed by *filters*, which are the processing components, and each filter is connected to the next by means of a *pipe*, which transfers data from the filter to its successor. Usually, the data are uniform and the tasks apply some sort of transformation on them, such as addition, modification or reduction of information. Although several descriptions exist for this style [11, 109, 63], the pattern presented here is suitable for the agent paradigm and has been adapted to be useful within the methodological context of Gaia. In particular, the components have been modelled as roles and agents, and their interactions as organisations.

- Restrictions. First, the overall task must be decomposable into independent sequential tasks. Second, the flow of information is restricted to be linear, sequential and only in one direction (no loops or feedback). Third, the processing speed is determined by the slowest filter, although the use of buffers in pipelines can alleviate this restriction to some extent. Finally, to avoid bandwidth and storage problems, the data transferred from stage to stage must be small.

- Consequences. The mechanism of coordination provided is rather simple and is not suitable for error management. This structure is flexible, since filters can be replaced or bypassed and new filters can be easily added.

- Implementation. The overall task of the system has to be decomposed into independent sequential tasks, with each assigned to one filter. The pipelines may be immersed in the communication layer, in which case they will not be directly associated to any agent of the system.

- Roles. Filters are obvious candidates to become roles. In addition, pipes are also modelled as roles since this highlights their existence within the structure. (The decision to join a filter and a pipe in a single agent can be postponed to the detailed design phase. Alternatively, pipes could be modelled as resources.) However, it should be noted that filters are *active* entities while pipes are *passive* ones. Filters are allowed to be sub-organisations themselves, but pipes are assumed to be primitive entities. For simplicity of the pattern, the roles of data source (the component which supplies data to the first pipe) and the data sink (the component to which the data to the last pipe is supplied) are not included. Figures 3.11 and 3.12 show templates of role schemata for the filter and pipe roles respectively.

- Structure. Let us denote with N the number of filters in the structure and with $Filter_i$ and $Pipe_j$ the filters and pipes ($1 \leq i \leq N$ and $1 \leq j \leq N+1$) respectively (note that the number of pipes is $N+1$). (Figure 3.13 depicts a pipeline for the case $N = 3$.) The

Role Schema:	$Filter_i$
Description:	Performs the process corresponding to stage i on the input data
Protocols and Activities:	<u>$ProcessData_i$</u>, GetInput, SupplyOutput, <u>SenseFlows</u>, <u>ChangeFlow</u>
Permissions:	changes *Data*, $flow_i$, *agreedFlow* reads $flow_j$

Responsibilities:

Liveness:

$Filter_i$ = (Process | AdjustFlow)^{w}

Process = GetInput.<u>$ProcessData_i$</u>.SupplyOutput

AdjustFlow = <u>SenseFlows</u> | <u>ChangeFlow</u>

Safety:

•true

FIGURE 3.11: The Filter role

Role Schema:	$Pipe_i$
Description:	Transfers data (from $Filter_{i-1}$) to $Filter_i$ using a buffer
Protocols and Activities:	<u>Fetch</u>, <u>Store</u>, <u>CheckOverflow</u>, GetInput, SupplyOutput
Permissions:	reads *Data*

Responsibilities:

Liveness:

$Pipe_i$ = (Transfer) ^{w}

Transfer=(GetInput.<u>Fetch</u>) |

(SupplyOutput.<u>Store</u>)

Safety:

•BufferOverflow = false

FIGURE 3.12: The Pipe role

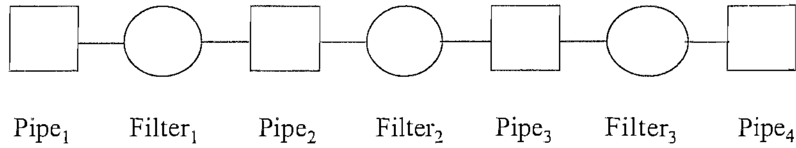

FIGURE 3.13: Topology of a pipeline structure

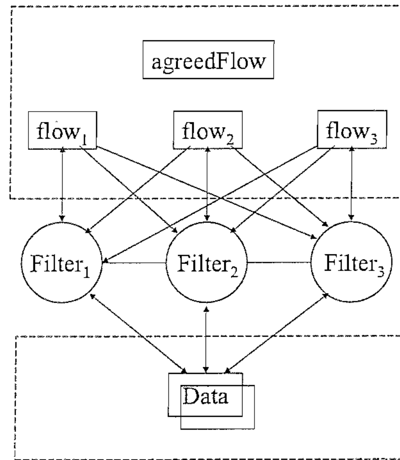

FIGURE 3.14: The environmental entities of the pipeline structure

structure is described by the following control regime (see Section 3.4.3.3):

$$\{peer(Pipe_i, Filter_i) \mid i = 1 \ldots N\} \;\bigcup\; \{peer(Filter_i, Pipe_{i+1}) \mid i = 1 \ldots N\}$$

(Each pipe interacts with the filter *to its right* and each filter interacts with the pipe *to its right.*)

- Environmental entities. We denote by $flow_i$ the flow rate at stage i, and by $agreedFlow$ the operation flow of the overall pipeline. These are real numbers and their rights of access are shown in Table 3.12. For example, the entity corresponding to the flow of stage i, $flow_i$, can be modified only by the role $Filter_i$, but can be sensed by all the other filters. Also, we denote by $Data$ the items that the pipeline precesses. In Figure 3.14, these

Entity	Type	Description	Modified by	Sensed by
$flow_i$	real number	flow rate at stage i	$Filter_i$	$Filter_j, j \neq i$
$agreedFlow$	real number	pipeline's operation flow	$\forall j\ Filter_j$	$\forall j\ Filter_j$
$Data$	application specific	the item processed	$\forall j\ Filter_j$	$\forall j\ Pipe_j$

TABLE 3.12: Environmental entities of a pipeline and their rights of access

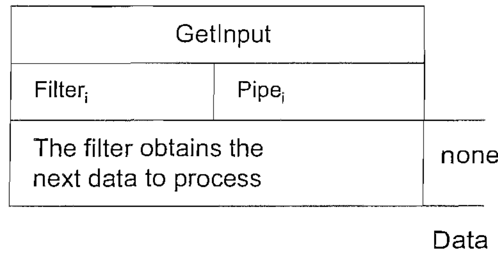

FIGURE 3.15: The GetInput protocol

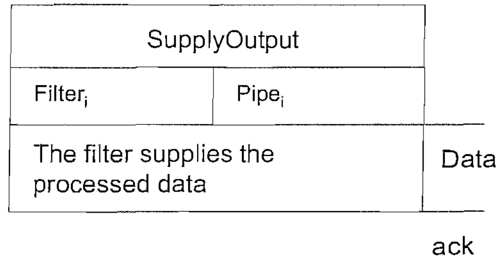

FIGURE 3.16: The SupplyOutput protocol

environmental entities have been depicted with boxes, and the rights of access with lines of one or two arrowheads, for sensing and modifying, respectively. The figure highlights the fact that these entities are not part of the roles by enclosing them in a dotted box.

- Dynamics. There are two main scenarios in the dynamics of a pipeline structure: *item processing* and *flow adjustment*. The former refers to the normal operation of the structure in which items are processed, while the latter deals with the way the agreed flow of operation is adjusted.

  The first scenario, item processing, relies mainly on the interaction of the roles in the structure. As shown in the filter and pipe schemata (Figures 3.11 and 3.12), the protocols involved in the operation of the organisation are $GetInput$ and $SupplyOutput$, described in Figure 3.15 and Figure 3.16, respectively. The typical operation of the structure at stage i is shown in Figure 3.17. First, the filter asks the pipe to its left for the next data using the $GetInput$ protocol (the filter confirms the correct receipt of the data); next, the filter

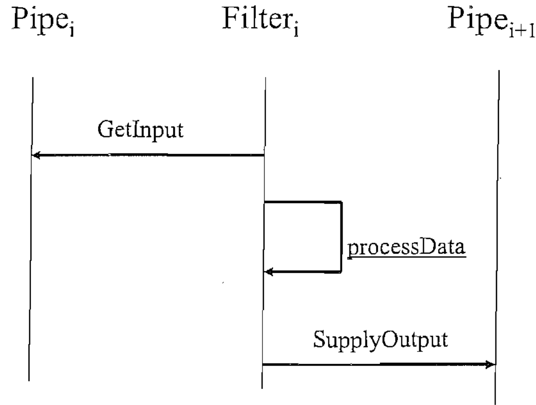

FIGURE 3.17: Dynamics of the pipeline structure

process the data; and finally, the filter asks the pipe to its right to store the processed data using the *SupplyOutput* protocol.

The second scenario, flow adjustment, relies on the capabilities of the roles to sense and affect the environment. In this case, it is assumed that the filters are continually sensing the environmental entities corresponding to the other filters' flows. When a filter wants to change the overall rate of flow ($agreedFlow$), it first modifies the environmental variable corresponding to its own flow of operation ($flow_i$, for the i corresponding to the stage). The new value of $flow_i$ can be less than the previous value (for example, if the filter is having problems with maintaining the current rate of operation), or greater (for example, when the filter has recovered from a previous performance downgrade). Later, when each filter senses all the filters' flows, the agreed flow is calculated as the minimum of all the filters' flows. This agreed flow is the flow at which they *agree* to *operate*, whereas a filter's flow is regarded as the flow at which the filter *wishes* to operate.

- Rules. The following are the organisational rules that control the operation of this structure.

  All the roles are played by at least one agent:

$$\forall i(\quad \exists a(\quad plays(a, Filter_i)))$$

$$\forall j(\quad \exists a(\quad plays(a, Pipe_j)))$$

  All the roles are played by at most one agent:

$$\forall i(\quad plays(a, Filter_i) \wedge plays(b, Filter_i) \Rightarrow a = b)$$
$$\forall j(\quad plays(a, Pipe_j) \wedge plays(b, Pipe_j) \Rightarrow a = b)$$

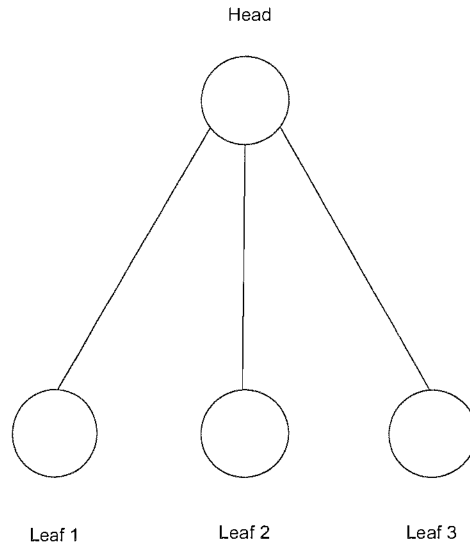

FIGURE 3.18: The simple hierarchy structure

The rate of process is the same for all the filters:

$$\forall i \in \{1, \ldots, N\}(\quad Filter_i.GetFlow() = agreedFlow)$$

Every filter delivers the data immediately after processing it:

$$\forall i \in \{1, \ldots, N\}(\forall d : Data(\quad terminated(Filter_i.ProcessData(d)) \Rightarrow \bigcirc initiated(SupplyOutput(d))))$$

Every filter fetches the next data immediately after delivering the previous data:

$$\forall d : Data(\exists d' : Data \; (terminated(SupplyOutput(d)) \Rightarrow \bigcirc initiated(GetInput(d'))))$$

3.6.3 The Simple hierarchy pattern

- Name: simple hierarchy.
- Alias: two-levels hierarchy.
- Context. According to the methodological process, before selecting a pattern the developer has already constructed the preliminary roles and the interactions models, and has also compiled the organisational rules. After selecting the appropriate pattern, the

developer must be able to complete the final roles and interactions models using the corresponding organisational structure.

- Problem. The problem in question is to find the organisational structure that best describes the system under development. In Gaia, the processes of the organisation are provided by the roles model, so what is missing is to define the topology and the control regime of the organisation. To this end, some characteristics of the problem have been identified. First, the overall goal can be naturally decomposed into a reasonable number of independent tasks arranged hierarchically. Second, although the tasks are independent, their execution require non-trivial coordination. Lastly, scalability is desired.

- Forces:

  - Coordination efficiency: medium.
  - Coupling: medium.
  - Subordination relationships: authority.
  - Topology complexity: simple.

- Solution. Hierarchies are one of the most used types of organisational structures [45], arguably because they are intuitive, simple in concept and relatively easy to implement. A simple hierarchy has the form of a two-level tree, as shown in Figure 3.18 (a simple hierarchy of three elements at the lower level). The top level contains a single element whose responsibility is to coordinate the activities of the lower-level elements or to consolidate the data provided by them. Usually, such coordination consists of the apex sending control orders to the lower-level elements and receiving from them the data they produced. For this to happen, there must exist an *authority* relationship from the apex to the lower-level elements (see Section 3.4.3.1). On the other hand, the consolidation of data performed by the apex usually involves some form of analysis, summarising, filtering or approximation. The consolidated information can then used by the apex for decision making.

  In a strict simple hierarchy, the only communication paths allowed are between the apex and the lower level elements, but in some applications, communication between the elements of the same level can be allowed to increase efficiency and robustness.

  Hierarchies are more useful in solving problems that can be decomposed into several independent tasks, in particular those that tend to grow in the number of involved tasks.

- Restrictions. First, the overall task must be decomposable into independent subtasks. Second, each one of these subtasks must not exceed the processing capabilities of the corresponding element, and the coordination and information consolidation tasks must not exceed the capabilities of the apex element. Third, direct communication between elements of the lower-level is usually not allowed.

- Consequences. This structure presents the following disadvantages: in case of frequent communication, bottlenecks may arise in the apex. Similarly, a failure in the operation of

Role Schema:	Head
Description:	Coordinates the activities of the elements at the lower level
Protocols and Activities:	Consolidate, DataAvailable, NextCommand, ChangeRate
Permissions:	changes *agreedRate*

Responsibilities:

Liveness:

Head =(ReceiveData ¦ NextCommand)^{w}

ReceiveData = wait.(DataAvailable | ChangeRate)

Safety:

- true

FIGURE 3.19: The Head role

the apex can have serious consequences in the operation of the whole structure. On the other hand, simple hierarchies present the following advantages: hierarchies have a good level of scalability in terms of the number of elements as well as the number of levels. Compared to the pipeline structure, a simple hierarchy improves efficiency and increases parallelism.

- Implementation. The most critical role of the structure is the *apex*, since it centralises the communication, performs the coordination tasks, and consolidates the information. For these reasons, it is important that the agent playing the role of the head is implemented with enough communication bandwidth, memory capacity and processing speed. Moreover, mechanisms of upgrade and backup are needed to recover from a possible failure in this role.

- Roles. There are essentially only two roles in a simple hierarchy: the role situated at the apex, which we call the *head*, and the role played by the elements at the lower-level, which we will call the *leaf*. Figure 3.19 shows the description of the head role and, as can be seen, the only activity of the head is to consolidate the received data. The head uses the protocols *DataAvailable*, *NextCommand* and *ChangeRate* to coordinate the behaviour of the *leaf* role, which is described in the schema of Figure 3.20. The figure shows that the behaviour of the leaf role is to wait for new commands, produce the corresponding data and inform of its availability.

- Structure. The following expression defines a simple hierarchy of N leaves.

Role Schema:	$Leaf_i$
Description:	Performs the data transformation corresponding to stage i
Protocols and Activities:	ProcessData$_i$, NextCommand, DataAvailable, changeRate
Permissions:	changes Rate$_i$ changes supplied *Data*
Responsibilities:	
Liveness:	$Leaf_i$ = (NextCommand.ProcessData$_i$.DataAvailable | changeRate)$^{\omega}$
Safety:	•true

FIGURE 3.20: The Leaf role

$$\{authority(Head, Leaf(i)) \mid i = 1 \ldots N\}$$

Thus, the participants set is:

$$\{Head\} \cup \{Leaf(i) \mid i = 1..N\} \; ;$$

and the control relationships set is:

$$\{authority\} \, .$$

- Environmental entities. We denote by $Rate_i$ the operation rate of $Leaf(i)$ and with $Data$ the information processed by the hierarchy. The former are real numbers, while the latter is a registry whose composition depends on the application.
- Dynamics. The main scenario of the operation of the simple hierarchy structure is depicted in Figure 3.21, which shows that the head requests a leaf to perform a specific task. As a result of this, the leaf produces data and notifies the head about its availability.
- Rules. The organisational rules governing this structure are the following.

  All the roles are played by at least one agent:

$$\exists a(\quad plays(a, Head))$$

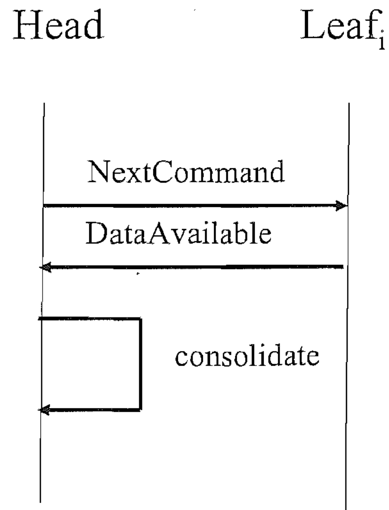

FIGURE 3.21: Main dynamics of the simple hierarchy structure

$$\forall j(\quad \exists a(\quad plays(a, Leaf_j)))$$

All the roles are played by at most one agent:

$$plays(a, Head) \wedge plays(b, Head) \Rightarrow a = b$$
$$\forall j(\quad plays(a, Leaf_j) \wedge plays(b, Leaf_j) \Rightarrow a = b)$$

3.6.4 The Marketplace pattern

One way to cope with task complexity is by *subdivision*, which consists in dividing the task into subtasks and then distributing them among several roles, or several agents playing the same role. This is useful only to some extent because, in general, the more subdivisions, the more complex coordinating the roles is. An approach to cope with task complexity without incrementing coordination complexity is by means of a *market system* [45] in which the task is accomplished by an independent entity that receives compensation for it. Normally, several entities are willing to perform the task, so a mechanism of selection is used. In this way, coordination is reduced just to the agreement of a contract.

- Name: Marketplace.
- Alias: Market or price system.

- Context. According to the Gaia process, before selecting a pattern the developer has already completed the roles and the interactions models, and has also compiled the organisational rules and defined the organisational structure (topology and control regime). After selecting the appropriate pattern, the developer must be ready to complete the final roles and interactions models.

- Problem. The problem in question is to find the organisational structure that best describes the system under development. In Gaia, the processes of the organisation are provided by the roles model, so what is missing is to define the topology and the control regime of the organisation. Additionally, the following characteristics of the problem have been identified: there is a strong belief that several tasks, or the main task, can be accomplished by independent agents following a metaphor of buying and selling products; it is likely that there will be a good number of agents willing to accomplish the task if they are appropriately paid; and it is not worthwhile to to accomplish the task by itself.

- Forces:

  - Coordination efficiency: high.
  - Coupling: low.
  - Subordination relationships: peer.
  - Topology complexity: simple.

- Solution. A possible solution to the problem is by using a marketplace organisational structure, which is an organisation that supports the sale of products (or services) by providing facilities for sellers and buyers. The set of facilities may vary in each marketplace, but common facilities for buyers include the discovery of purchase opportunities, mechanisms to select the best selling option and payment channels. For sellers, the marketplace usually offers marketing presence and payment channels. An additional benefit of marketplaces is that they provide buyer and seller trust in that the information provided there is reliable, and that the critical processes, such as payments, are confidential and secure.

  In order to provide these facilities, there exist one or several special agents, or *facilitators*, in the marketplace whose job is to perform common activities, thus freeing sellers and buyers from the burden of implementing them. The most important of these facilitators — which we have called the *Marketer* — communicates directly with sellers and buyers, matching a buyer with a seller, or acting as a communication channel between them. The Marketer frees the buyers from the burden of maintaining a directory of all possible sellers and vice versa. Additionally, a facilitator — here called the *Guard* — is needed to keep track of all the buyers and sellers that enter the marketplace, partially to maintain a directory of all the participants, but most importantly, to check that the new participants comply with the corresponding marketplace rules. Apart from this, in some marketplaces the mechanism to select the best selling option might be so complex that another facilitator is required. In this pattern we call such a facilitator the *Auctioneer* because the selection

process usually consists of some form of auction, but some marketplaces may require more general negotiation techniques.

- Restrictions. First, the task must be suitable to be accomplished by an external entity; this leaves out tasks that require much knowledge of the internal structure of the organisation. Second, the only control over the independent entity is through the price of the contract. Third, there must be at least one agent willing to accomplish the task; in general, the more the better. Finally, although no authority relation must exist from the facilitators to the buyers and sellers, at least a relationship of *trust* is necessary.

- Consequences. The disadvantages of using a marketplace organisation are: there is a risk that no buyer is willing to perform the task or that the final price is too high; since a marketplace is essentially a competitive organisation, the final price is difficult to predict; finally, buyers cannot impose explicit control on sellers. The advantages are: through competition, buyers tend to obtain a better price; and the coordination complexity is very low.

- Implementation. The implementation of a marketplace includes two types of activities: implementing the core of the system, the facilitators; and implementing the participants, the sellers and buyers. The former is carried out by the administrators or owners of the marketplace and consists mainly in developing the following critical parts of the system: a reliable communications infrastructure; a secure payment facility; a set of organisational rules that allow competition but without compromising the integrity of the system; a mechanism to allow new participants to have access to the system; and mechanisms and policies for offer selection. Since each of these parts has in itself been well studied in other areas of research, and there are in fact developments that can be used or adapted, the implementation of marketplaces need not begin from scratch. For example, communication infrastructure is provided by agent-oriented platforms such as JADE [70] or ZEUS [120], while the mechanisms for obtaining a best offer are covered in the study of auctions. The activities involved in implementing the participants are more closely related to a specific application and the particular needs of the owners. Also, although there is some freedom to select the best way to implement the participants, the resulting agents must comply with the system in terms of communication protocols and observance of the operational rules.

- Roles. We distribute the functionality of marketplaces into six roles, in accordance with the main tasks. Figure 3.22 and Figure 3.23 show the roles corresponding to the participants, *Seller* and *Buyer*, respectively. The roles corresponding to the facilitators, *Marketer*, *Auctioneer*, *PaymentSystem* and *Guard* are shown in Figure 3.24, Figure 3.25, Figure 3.26 and Figure 3.27, respectively.

- Structure. Figure 3.28 shows the topology of this organisational pattern. Note that all the control relationships between the nodes are *peer* relationships, which are derived from inter-role *trust* relationships. The following expression defines the organisational structure

Role Schema:	Buyer
Description:	Acquires a product or a service
Protocols and Activities:	BrowseOportunities, RequestProduct, ReceiveDeal, AcceptDeal, Pay, ReceiveProduct
Permissions:	reads Product, Contract, Offer produces Request
Responsibilities:	
Liveness:	Buyer=(BrowseOportunities . RequestProduct . ReceiveDeal . AcceptDeal . Pay . ReceiveProduct)
Safety:	Contract.payment <= balance

FIGURE 3.22: Roles in a marketplace, part 1

Role Schema:	Seller
Description:	Provides a product or a service
Protocols and Activities:	ReceiveRequest, ProposeOffer, ReceiveContract, AcceptContract, ReceivePayment, DeliverProduct
Permissions:	reads Product, Contract, Request produces Offer
Responsibilities:	
Liveness:	Seller=(ReceiveRequest . ProposeOffer . ReceiveContract. AcceptContract . ReceivePayment . DeliverProduct)^{w}
Safety:	stock(Contract.IdProduct, Contract.DispatchDay) > 0; dateOfDispatch <= Contract.DispatchDay

FIGURE 3.23: Roles in a marketplace, part 2

Role Schema:	Marketeer
Description:	Serves as a link between a seller and the potential buyers
Protocols and Activities:	AttendToProductRequest, RequestToPotentialSellers, ReceiveBestOffer, SendContracts, ReceiveContractAcceptances, InstructPayment
Permissions:	reads Offer, Request changes Product, Contract
Responsibilities:	
Liveness:	Marketer=(AttendToProductRequest . RequestToPotentialSellers . ReceiveBestOffer . SendContracts. ReceiveContractAcceptances. InstructPayment)^{w}
Safety:	exists(Request.ProductId) Offer.ProductId = Request.ProductId

FIGURE 3.24: Roles in a marketplace, part 3

Role Schema:	Auctioneer
Description:	Selects the best offer
Protocols and Activities:	ReceiveOffers, SelectBestOffer, NotifyBestOffer
Permissions:	reads Offer
Responsibilities:	
Liveness:	Auctioneer=(ReceiveOffers . SelectBestOffer . NotifyBestOffer)^{w}
Safety:	.

FIGURE 3.25: Roles in a marketplace, part 4

Role Schema:	Guard
Description:	Checks that new participants comply with the requirements
Protocols and Activities:	ReceiveNewParticipant, CheckCompliance
Permissions:	produces PermissionToOperate
Responsibilities:	
Liveness:	Guard=(ReceiveNewParticipant . CheckCompliance)^{w}
Safety:	PermissionToOperate.State = true

FIGURE 3.26: Roles in a marketplace, part 5

Role Schema:	PaymentSystem
Description:	Receives payments of sales
Protocols and Activities:	ReceivePayment, NotifyPayment
Permissions:	reads Contract
Responsibilities:	
Liveness:	PaymentSystem=(ReceivePayment . NotifyPayment)^{w}
Safety:	

FIGURE 3.27: Roles in a marketplace, part 6

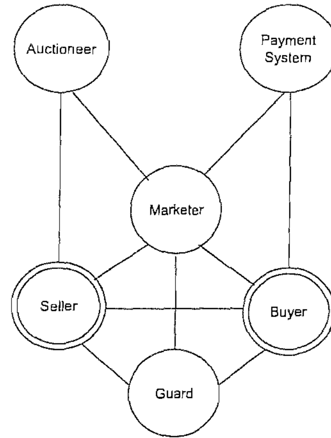

FIGURE 3.28: Topology of a market

Entity	Type	Description	Modified by	Read by
Product	registry	good or service to sell	Marketer	Buyer, Seller
Contract	registry	deal of transaction	Marketer	Buyer, Seller, PaymentSystem
Offer	registry	bid	Seller	Auctioneer, Marketer, Buyer
Request	registry	product required	Buyer	Seller, Marketer

TABLE 3.13: Environmental entities of a Marketplace and their rights of access

of the marketplace pattern. In this figure, we have denoted with double circle those roles whose cardinality can be greater than one.

$$\{peer(Seller, Buyer), peer(Seller, Marketer), peer(Seller, PaymentSystem)\} \cup$$

$$\{peer(Buyer, Marketer), peer(Buyer, Guard), peer(Marketer, Guard)\} \cup$$

$$\{peer(PaymentSystem, Marketer), peer(PaymentSystem, Buyer)\} \cup$$

$$\{peer(Auctioneer, Seller), peer(Auctioneer, Marketer)\}\,.$$

- Environmental entities. The environmental entities of the Marketplace pattern are: *Product*, *Contract*, *Offer*, and *Request*. A *Product* represents the good or service involved in a transaction, a *Contract* is the deal to which the seller and the buyer of a transaction agree to adhere, an *Offer* is the proposal (usually a price) a buyer makes to buy a product, and a *Request* consists of the requirements for a product that a buyer wants to buy. These entities are data structures whose composition depends on the specific application. Table 3.13 summarises the rights of access that the roles have on these environmental entities.

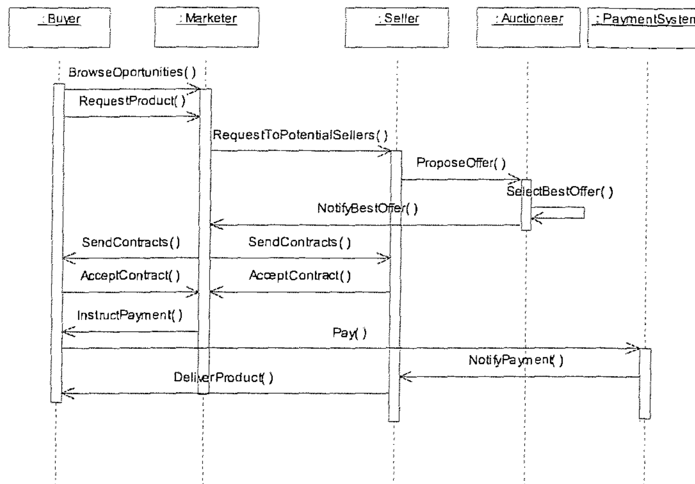

FIGURE 3.29: The Sale scenario of the marketplace structure

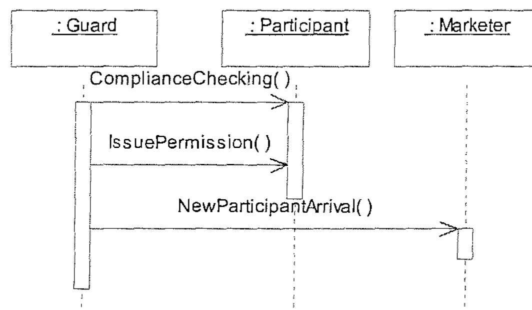

FIGURE 3.30: The Entrance scenario of the marketplace structure

- Dynamics. There are two main scenarios in the operation of a marketplace, one corresponding to the sale process and the other to the entrance of a new participant to the system. Figure 3.29 shows how the roles interact to achieve a sale: a $Buyer$ sends a product request to the $Marketer$, who contacts all the potential sellers. The sellers interested in fulfilling the request send their offers to the $Auctioneer$, who selects the best offer. After that, the $Marketer$ prepares the contract and sends it to the $Buyer$ and $Seller$ for their approval. Finally, the payment and product delivery are carried out. Figure 3.30 corresponds to the entrance of a new seller or buyer. In this simple scenario, the $Guard$ checks that the new participant complies with the requirements of the system, in which case a permission to operate is issued, and the $Marketer$ is notified of the entrance.

- Rules. The organisational rules that govern this marketplace are the following.

  The buyer and the seller cannot be played by the same agent:

$$plays(a, Buyer) \wedge plays(b, Seller) \Rightarrow a \neq b$$

The seller and the auctioneer must be played by different agents:

$$plays(a, Seller) \wedge plays(b, Auctioneer) \Rightarrow a \neq b$$

A product is not delivered before the payment has been received:

$$terminated(NotifyPayment)\ \mathcal{B}\ initiated(DeliverProduct)$$

Payments are not made before contract acceptance:

$$terminated(AcceptContract(Buyer)) \wedge terminated(AcceptContract(Seller))\ \mathcal{B}\ initiated(DeliverProduct)$$

The marketer is not informed about a new participant before the participant has obtained permission:

$$terminated(IssuePermission)\ \mathcal{B}\ initiated(NewParticipantArrival)$$

Only one offer is selected in each auction:

$$card(NotifyBestOffer) = 1$$

When we compare the marketplace pattern against the FM market [106], a practical agent-based market, we find several differences and similarities. Regarding the roles in the structure, FM employs eight roles, of which the *Auctioneer*, the *Buyer*, and the *Seller* coincide in name and function with the corresponding roles in the marketplace pattern. FM includes two roles for tasks of admitting participants, namely the *Buyer Admitter* and the *Seller Admitter*, whereas the marketplace encompasses both functionalities in one role, the *Guard*. The rest of the FM roles, the *Boss*, the *Seller Manager* and the *Buyer Manager*, enact the functionality represented in the pattern by the *Marketer* and the *PaymentSystem*. In particular, one of the functions of the *Seller Manager* is to provide a facility for payments, which in the pattern is carried out by the *PaymentSystem*. While this provides a good means for comparison with one specific market, the unavailability of suitable documentation prevents us from make a more detailed analysis or from a broader analysis against other agent-based markets, such as Kasbah [17] and TAC [113].

3.6.5 Selecting patterns

Even for applications involving just a few roles and protocols, it is sometimes difficult to recognise if their organisational structure is *similar* to one of those typical structures such as group, hierarchy or marketplace. This arises because it is difficult for humans to *visualise* the whole of

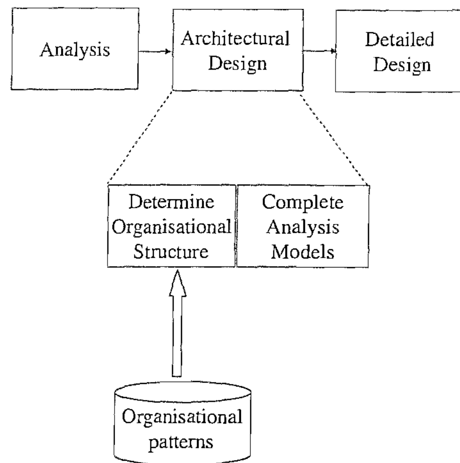

FIGURE 3.31: The role of organisational patterns in the methodology

a structure except for the simplest cases, or if the elements of the structure are represented in a form with which we are not familiar. For this reason, it might be difficult to select — among a set of organisational patterns — the pattern that best matches a given structure.

3.6.6 Summary

As mentioned previously, organisational patterns show their real benefit when used as part of an agent-oriented software methodology. The details of using organisational patterns in a Gaia-like methodology were described in this chapter, and are summarised below.

Figure 3.31 illustrates the role of organisational patterns in the methodological process. As can be observed, organisational patterns are used during the architectural design phase, after the analysis has been done. The analysis provides the preliminary versions of the roles and interaction models, as well as of the organisational rules. From this information, an outline of the organisational structure of the system is obtained.

However, the selection of the definitive organisational structure of a system depends not only on the preliminary models, but also on other factors such as the complexity of computation of the roles, the complexity of coordination of potential organisational structures, the real-world organisation that is being modelled, and the support offered for observance of the organisational rules.

It is precisely in this process of selection that a catalogue of organisational patterns plays an important role, by providing pre-defined options from which an appropriate organisational structure can be selected. Such a selection is carried out basically by matching the characteristics of the system-to-be with the characteristics provided by the patterns. In addition, given their high

re-usability, organisational patterns usually contain more information than is found in organisational structures at that stage of development.

As a result of selecting an specific organisational pattern, the architectural design is completed by incorporating the elements of the pattern into the analysis models. More specifically, the preliminary roles and interaction models, and the analysis' organisational rules, are updated with the roles, interactions and control relationships of the pattern, and the organisational rules of the pattern are added.

Finally, the organisational structure and the architectural design models form the inputs to the detailed design, which produces a list of roles enacted by each agent in the system and a list of services provided by each agent.

3.7 Related work and conclusions

The patterns presented here are intended to be used during the methodological process outlined by Zambonelli et al. [133], in which the importance of a set of organisational patterns is stated but no such set is presented.

Patterns are extensively used to facilitate the development of software systems; in the agent-oriented approach they have been employed to design multiple aspects of an application. Some examples of agent-based methodologies that include the use of patterns in their processes are Tropos [81], Kendall's methodology [79] and PASSI [20]. As part of the Tropos methodology, Kolp et al. present a set of patterns in [81], in which patterns (called *styles*) are used to describe the general architecture of a system under construction. Although there are similarities with our work, we include organisational rules and classify structures based on topology and control regime (or task decomposition), as opposed to the classification based on functionality used in Tropos.

Kendall [79] also includes a catalogue of patterns as a part of a technique to analyse and design agent-based systems. The patterns in that catalogue are more general than those presented here, since they include not only interactions but also the roles themselves (it should be noted that the concept of *role* there comes from role theory and is not identical to the concept used here). Since there is no reference to organisational abstractions, that work cannot be directly used in the methodological process we use, but perhaps the structure of those patterns may be used as a base to populate the set of patterns proposed here.

Immersed in the PASSI methodology, Cossentino et al. present in [20] the design of a particular type of agent pattern. They define a pattern as consisting of a model and implementation code. The model includes two parts: structure and behaviour. Structural patterns are classified into: action patterns, which represent the functionality of the system; behaviour patterns, which can be viewed as a collection of actions; component patterns, which encompass the structure of an agent and its tasks; and service patterns, which describe the collaboration between two or more

agents. Implementation code is available for two agent platforms, namely, JADE and FIPA-OS. As can be noted from this brief description, the concept of organisation is not explicitly addressed in their work.

Other patterns in the agent literature do not use a specific methodology. For instance, Aridor and Lange [6] present a catalogue that covers different aspects of an application based on mobile agents, travelling (management of the movements of a mobile agent), task (task decomposition and assignment), and interaction (locating agents and facilitating their interaction), but these are appropriate only for mobile-agent systems, and are *object-based* rather than role-based. Lind [84] proposes a structure of a pattern catalogue in which the work presented here may fit in the *Society* section, but it is not always clear how to apply the general-purpose patterns within a specific methodology. This is also true for [26], in which Deugo et al. present a set of coordination patterns that are not embedded in a methodology process. Their usage is more complicated due to the fact that there is no separation of the different type of patterns, for example, coordination patterns and task delegation patterns. A similar set of patterns is presented by Hayden et al. [62] but this focuses on defining how a goal assigned to a particular agent is fulfilled by interacting with other agents. Finally, Silva and Delgado [114] present an agent pattern that provides distribution, security and persistence transparency. This does not suit our purposes because it focuses on access to a single agent rather than considering an organisation of them.

Although several agent-oriented methodologies have recently been proposed, none of them is mature enough to develop commercial and industrial applications. One step towards achieving mature methodologies is to enhance existing ones with the inclusion of software engineering best practices, such as the use of patterns in key parts of the design process. In this chapter we have presented a framework in which organisational patterns may be developed to model the organisational structure of software applications. Also included are three patterns corresponding to representative structures. No framework or set of patterns like these have been proposed before.

Specifically, the contributions of this chapter to the state of the art of agent-oriented software engineering are the following.

- Definition of a language for the expression of organisational rules, namely *LEVOR*. The manipulability of LEVOR also makes it suitable for the analysis and evaluation of organisational rules at run-time.
- Construction of a formal model for organisational structures. This model, although simple, is expressive enough to formally describe organisational structures, and link them with their corresponding graphical representation.
- Definition of a layout for the description of organisational patterns. Such layout facilitates the understanding and use of the patterns in a catalogue.

- Construction of an initial catalogue of patterns. This catalogue uses the layout referred above and includes the patterns corresponding to some representative organisational structures.

The work presented here provides four distinct benefits. First, it extends Gaia, which is one of the most used methodologies, since the exploitation of organisational patterns is an integral part of its process. Second, it increases the accessibility of the methodology, in that the inclusion of patterns makes the methodology easier to use, especially by non-expert users. Third, it helps to reduce development time since developers may reuse the models to avoid building their applications from scratch. Finally, it provides a basis on which further patterns can be developed and improvements can be discussed.

It should be noted that although some patterns are very simple in concept, their usefulness is twofold: they explicitly state the structure a system must conform to; and they serve as a basis for designing complex applications, since, arguably, most real applications can be described by a composition of several simpler structures.

Part of the work involved in creating the catalogue of patterns has also dealt with the characterisation of organisational structures and organisational rules — which are the most important parts of the patterns — and the languages to describe them. Although these problems have been explored before by other researchers [134, 44], our work includes formal characterisations and languages. These characterisations and languages are independent of the patterns and can be used on their own for other more general purposes, such as the analysis and design of agent-based systems.

However, the catalogue presented in this chapter has some limitations in terms of maturity. A catalogue is an evolving project that improves with the participation of a community of users, serving as a repository of the expertise of a community. It gains maturity through the participation of a community of users in activities such as:

- adding new patterns to the catalogue;
- adding variants to existing patterns in the catalogue;
- providing feedback about the utility of a pattern;
- proposing generalisations of a pattern; and
- increasing the level of detail of a pattern.

As long as the population of the catalogue increases, new needs arise. For example, with a large number of patterns in the catalogue, it is necessary to have a *comparison table* that summarises the characteristics of the pattens. Also, in those cases in which the architectural design is supported by a software tool (CASE tool), it might be desirable to feed that tool with the description of the selected pattern, in order to speed up the design process.

Chapter 4

Modelling the internal structure of an agent

4.1 Introduction

As discussed in Chapter 2, the design of a multi-agent system consists of two main parts: the design of the interactions between the agents (in which agents are essentially viewed as black boxes), and the design of the internal composition of each agent. While we addressed the former in Chapter 3, in this chapter we consider the latter.

Although for each application the agents can be designed following tailor-made techniques, the existence of standardised, well-defined and comprehensive methods for designing the internal structure of agents is important for engineering solutions in a repeatable and controlled form. For these reasons, we argue that modelling the internal structure of agents should be a mandatory phase in every agent-oriented software methodology.

In spite of this, the phase of modelling the internal structure of agents is not included in most current methodologies or, when included, has some serious drawbacks. Regarding the former, some methodologies consider this phase as part of the detailed implementation and thus out of their scope [134]. This leaves practitioners with the problem of how to complete the specification and, eventually, the implementation of the system, which can be a significant problem particularly for novice practitioners, and may discourage the widespread use of such methodologies. When this phase *is* included in the methodology, the internal representations are tied to specific — sometimes proprietary — models [15, 78, 55]. In the case of proprietary models, apart from forcing their use, there is no connection between them and those models that the research community has constructed in the field of *agent architectures*. Thus, they provide no insight into how to incorporate well known architectures, or how the proprietary models compare to them.

Rather than creating new and isolated models, or architectures, this thesis strongly relies on the employment of well known agent architectures. Specifically, in this chapter we explore

the use of existing well known architectures to support the design of agents. Although this approach is very natural, taking it into practice is difficult, mainly because agent architectures have not always been envisaged from a software engineering point of view. Basically, the use of agent architectures in software development presents a problem of mismatch: on the one hand, most current methodologies do not incorporate the expertise achieved in the study of agent architectures, and on the other hand, the specification of most agent architectures is rarely oriented towards software engineering practitioners

Regarding the difficulties in using agent architectures for modelling the internal structure of agents, we can mention the following. First, most architecture descriptions are available only at a high level of abstraction, or are not detailed enough, or are difficult to understand by non-agent experts. Second, there is a large number of architectures and it is difficult to select one between them for the implementation of a specific application.

In order to facilitate the use of well known agent architectures in the design of agents, we propose the construction of the three following *artefacts*.

- A catalogue of representations corresponding to some of the most used architectures, together with a selection criterion. The representations include design specifications and descriptions of the requirements for using the specifications. The descriptions of these elements must be such that they can be understood even for practitioners not proficient in agent technology.
- A technique to obtain a detailed design of an agent from the specification produced by a *high-level* methodology (such as Gaia [134]), using the representations mentioned above.
- A procedure to guide the development of a representation for any architecture not considered previously.

The combined use of these artefacts can be helpful in several situations that arise during the development of an agent. Once the requirements for the agent have been determined, the catalogue can be searched for a representation that appropriately models the agent, and then the technique provided can be used to obtain a detailed design. In this way, developers do not begin each design from scratch, nor do they have to go into all the details of agent architectures to determine which is the most appropriate for a particular agent. However, if the catalogue does not contain any appropriate representation, the procedure provided can be used to obtain a representation for a different architecture, and this representation added to the catalogue for its eventual re-utilisation.

For these reasons, artefacts like these facilitate the design of the internal structure of agents. However, since no artefacts like these exist to date, we need to establish methods for their construction and use. The chapter is organised in the following way. In Section 4.2 we describe our approach to modelling the internal structure of agents, in particular we justify the use of *software patterns* to represent such internal structures. Then, in Sections 4.3, 4.4 and 4.5, the patterns

for some selected agent architectures are presented. After this, in Section 4.6, methodological guidelines are provided for obtaining the corresponding pattern for any other architecture. Finally, Section 4.7 contains related work, and Section 4.8 contains our conclusions.

4.2 Internal representation of agents

The main goal of this section is to provide a framework in which a catalogue of representations may be developed. The aim of each representation is to model the internal structure of a family of agents, according to the principles established by a known agent architecture. However, as mentioned previously, several problems arise when using agent architectures in a methodological approach, mainly because most agent architectures are not described in a form that is useful for software developers. In particular, some descriptions are difficult to understand by non-agent experts, because they contain vocabulary and concepts that are unfamiliar and intricate. To alleviate this, we choose to describe the internal structure of agents by means of *software patterns* [11] (as used in Chapter 3, although for a slightly different purpose), since they are artefacts with which most software developers are familiar and, at the same time, are a powerful abstraction to represent families of solutions. Moreover, the object-oriented approach is suitable to serve as a base for the description of these patterns, since it is one of the most used design techniques and does not necessarily force an object-oriented implementation.

In summary, our approach consists of the creation of a catalogue of software patterns for selected architectures. The patterns include a description of their components, and the way in which they interact, as well as a description of the situations in which they are applicable. We evaluate the results of this chapter by means of a case study, which is presented in Chapter 6.

In contrast to other approaches that leave the user with the work of customising a general architecture to meet their specific applications [78, 96], our work provides concrete designs that correspond to well known architectures. Not only does this work include static descriptions of the internal structure of agents, it also addresses the dynamics of their main scenarios.

4.2.1 Obtaining a detailed design from a high-level design

High-level design of multi-agent systems focuses on organisational modelling, leaving agent modelling unconsidered, but a complete methodological process must provide links between these two aspects. Although architectural patterns, such as those presented in this chapter, are helpful in creating such links, there is still a gap between the results obtained by a high-level design (one that does not consider the detailed modelling of the internal structure of the agents), and the information required to use a specific agent architecture. This gap is a consequence of the difference in concepts and abstractions used in the design and the architecture. To bridge this gap, we have included, after each pattern in the catalogue, an explanation of how to obtain the information required by the pattern, from the specification given in a high-level design.

However, since this specification varies from methodology to methodology, we first need to establish what the output of the high-level design is.

In order to do this, we use the Gaia [134] methodology as representative of a methodology which produces a high-level design but does not consider a detailed agent design. Under this assumption, the results obtained by the high-level design are the following. (Each feature is described in more detail elsewhere in this thesis, so here we just provide a brief description and refer to the corresponding sections.)

The overall system is modelled through a set of *agents*, each of which plays one or more *roles*. Such roles interact according to *control relationships*, governed by *organisational rules*. Control relationships describe how two roles are related in terms of subordination; for example, control relationships might be *authority* and *peer*. In this view, organisational rules are thus restrictions about how roles and agents can interact; they may also involve other entities in the system, such as environmental entities.

Each role is described by *activities*, *protocols*, *permissions* to use resources, *responsibilities* and *services*, where:

- activities are tasks that a role can carry out alone, without interacting with other roles;
- protocols are patterns of interaction between the roles and consist of initiator, collaborators, input parameters and output parameters;
- permissions express the rights a role has to access the entities of the environment;
- responsibilities encompass the behaviour of the agent and are divided into *liveness* and *safety*, the former specifying the behaviour the agent pursues, and the latter specifying the conditions that must keep invariant through the life of the agent; and
- a service is a single coherent block of activity in which the agent will be engaged, and consists of pre-conditions, post-conditions, inputs, and outputs.

For each of the architectural patterns presented below, we have included a procedure to move from the design obtained by the methodology, to the information required by the pattern. Before presenting the patterns, however, we establish the layout used to describe them.

4.2.2 Pattern layout

As was discussed in Section 3.5.2, the way patterns are described, or their layout, is important to facilitates their understanding and their use. In order to describe architectural patterns, we use the layout utilised by Buschmann et al. [11], but only slightly modified to leave out the sections corresponding to alternative names of a pattern, its variants and references to closely related patterns, since they are useless for the patterns presented below. (Note that this layout

was used by Buschmann et al. to describe *design* patterns, of which these architectural patterns are a particular case.) In this way, the pattern layout we use consists of the following sections.

- Name: a unique identifier of the pattern.
- Context: a description of the situation in which the agent architecture is applicable.
- Problem: the problem addressed by the pattern.
- Solution: steps to follow in order to solve the problem, based on what is stated in the structure and dynamics sections.
- Known uses: real-world or experimental applications in which the architecture has been used.
- Structure: the structural aspects of the internal composition of an agent (according to the architecture), represented by means of a class diagram.
- Dynamics: the way internal components of an agent interact to accomplish its behaviour, divided into scenarios (meaningful parts of functionality).
- Implementation: guidelines for implementing the pattern.
- Consequences: benefits and limitations of using the architecture.
- Example: example to clarify the exposition or use of the pattern.

As can be observed, this layout considers the Context-Problem-Solution metaphor discussed in Section 3.5.2.2, which states that the essence of a pattern relies on the relationship between the problem, the situations in which it occurs (context), and its solution. Apart from the sections related to this metaphor (*context*, *problem* and *solution*), the layout also includes sections to identify the pattern(*name*), to describe the static and dynamic aspects of the solution (*structure* and *dynamics*, respectively), and to facilitate the use of the pattern (*known uses*, *implementation*, *consequences* and *example*).

4.3 The Subsumption architectural pattern

Using the layout presented above, in the following we describe the patterns corresponding to three well known agent architectures, namely the subsumption, the dMARS and the Touring-Machines architectures. For each pattern, we include a procedure to use it in the context of a methodological process. This section considers the architectural pattern corresponding to the subsumption architecture. (A description of the subsumption architecture was given in Section 2.3.1.)

4.3.1 The subsumption architecture

The subsumption architecture is a well-known reactive architecture that has inspired several other reactive and hybrid architectures. Although simple in concept, the subsumption architecture contains vocabulary and notions not commonly found in mainstream software engineering and, consequently, difficult to assimilate — in a first attempt — by a typical software developer, such as the concepts of *behaviour* and *inhibition relationship*. In this sense, the pattern presented below can act as a self-contained tool that developers can use for facilitating the construction of agents, since the operation of the architecture is put into common software engineering terms.

4.3.2 Pattern description

In order to clarify the description of the subsumption pattern, we make use of the following example [128, p51].

> "The objective is to explore a distant planet or, more concretely, to collect samples of a particular type of precious rock. The location of the rock samples is unknown in advance but they are typically clustered in certain spots. A number of autonomous vehicles are available that can drive around the planet collecting samples and later re-enter the mother spacecraft to go back to earth. There is no detailed map of the planet available, although it is known that the terrain is full of obstacles — hills, valleys, etc. — which prevent the vehicles from exchanging any communication".

The solution to this problem is based on two considerations [128]. First, since the terrain does not allow direct communication, vehicles communicate by means of *crumbs*. Once a vehicle has detected a cluster of samples, it shares its knowledge with other vehicles by repeatedly dropping two crumbs along its path to the mother spacecraft. In this way, a vehicle simply has to follow a track of crumbs to reach a cluster of samples. However, on its path from the mother spacecraft to a cluster, each vehicle picks up only one crumb. This allows both the persistence of the pathway for future vehicles, and the clearing of pathways in which the samples have been exhausted. The second consideration is the use of a signal emanating from the mother spacecraft together with its *gradient* field. To return to the mother spacecraft, a vehicle must follow the direction with highest gradient, while a vehicle intending to get away from the mother spacecraft has to move towards the direction with the lowest gradient. The use of the *subsumption architecture* to model the structure of such an agent is justified mainly because it is difficult to maintain a symbolic model of the environment, and because quick responses are required. This pattern is described in the following.

NAME Subsumption.

CONTEXT A software developer has designed a multi-agent system at the macro level, which means mainly the identification of agents, their responsibilities, and their interactions. The next step is to model the internal structure of the identified agents. It has also been determined that one agent requires reactive behaviour, so the subsumption architecture is suitable for describing its internal structure. The developer might not be an expert in agent architectures, so it would be desirable to have a mechanism that hides general aspects and lets the developer focus on application-specific details. For example, the developer does not need to be aware of details of the algorithm of *action-selection* (to select one action, for execution, from a set of proposed actions).

PROBLEM The developer needs to specify the implementation of an identified agent, and has already obtained the main characteristics of the agent, mainly in terms of its behaviour. It is clear that the agent does not need to maintain a complex mental state and, at the same time, needs to yield opportune responses to fast changes in the environment.

SOLUTION In order to utilise this architecture, the steps below must be followed.

1. Describe the environment as a set of states that can be recognised by the agent (through its perception function, commonly referred to as its *see* function). Note that this set represents what the agent is actually capable of perceiving from the environment.
2. Decompose the functionality of the agent into behaviours.
3. For each behaviour, determine its perceptual input (from the set of environment states), and describe its task.
4. Establish the inhibition relation, by assigning priorities to behaviours: more abstract behaviours have higher priority while basic behaviours have lower priority. More formally, the inhibition relation is a total ordering relation (transitive, irreflexive, and antisymmetric) on the set of behaviours.
5. With the components obtained in the previous steps, complete the classes of the class diagram shown in Figure 4.1 (which is explained in the STRUCTURE section) to obtain a design specification for the agent.

KNOWN USES The subsumption architecture has been applied to the control of robots that operate in unconstrained dynamic real-world environments. These robots wander in their environments avoiding collisions with other robots, objects and humans [10].

STRUCTURE Since the subsumption architecture is relatively simple, and involves only a small number of classes, we omit the use of packages to describe its structure. The main classes of the pattern and their relationships are shown in Figure 4.1. The control resides in the *SubsumptionController* class, which senses the environment through the *PerceptualInterface* class, affects the environment by means of the *EffectorInterface* class, and consults the class *Inhibitor* about the priority of the behaviours.

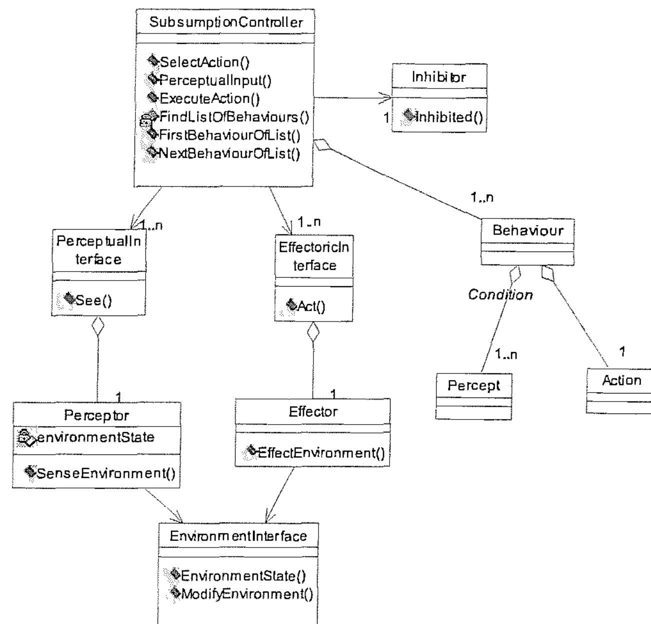

FIGURE 4.1: Class diagram for subsumption architecture

With the purpose of adapting to different situations, it is permitted for an agent to have more than one perceptual or effectoric interfaces. The environment itself is sensed by the *EnvironmentInterface*, but what the agent actually perceives is obtained from the *PerceptualInterface*. Similarly, the *EffectoricInterface* is the class that abstracts the actions that affect the environment. For example, in a FIPA-type agent ([35]) the *EnvironmentInterface* would contain the communications infrastructure, the *Perceptor* would encapsulate the recognition of the Agent Communication Language, and the *PerceptualInterface* would manipulate the information at the application level.

DYNAMICS A high level view of operation of the subsumption architecture is depicted in Figure 4.2, which uses a sequence diagram to show the participating classes, the involved methods and their order of execution. As can be observed, the control module is continually sensing the environment and matching its state to the perceptual conditions of behaviours. Among all these behaviours, the control module chooses one, and then the corresponding action is performed.

IMPLEMENTATION The algorithm for selecting a behaviour (and thus an action) among those matching the current state of the environment is quite simple to implement. First, the set of all the behaviours fired by the state is obtained. Second, the behaviour selected is the one with the minimum priority, or none if it does not exist. Finally, the corresponding action is returned.

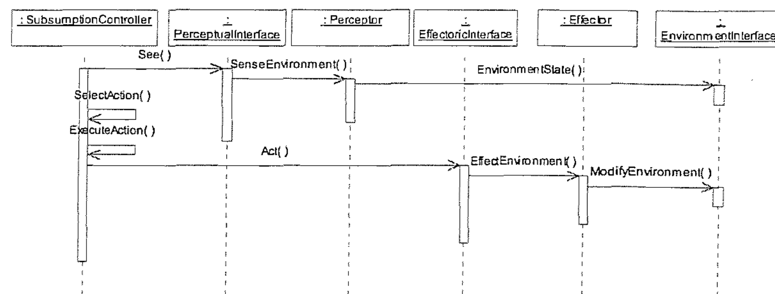

FIGURE 4.2: Dynamics for subsumption architecture

All the classes may belong to the same executable, although a major decoupling may be achieved if the *Effector* and *Perceptor* classes are separated. The *EnvironmentInterface* class may even have its own thread of execution if continuous examination of the environment is required.

CONSEQUENCES The disadvantages and limitations of this architecture are as follows. First, the subsumption architecture is behaviour-oriented while most current design methodologies are goal-oriented. This mismatch imposes some constraints on the functionality of some agents, particularly on agents with multiple goals. In addition, some effort on translating from goals to behaviours must be exerted during the design. Second, the absence of explicit representations of goals (like *intentions* in BDI architectures) makes the specification of pro-active behaviour difficult. Thus, this architecture is more suitable for agents that act in response to a request than pro-active ones.

On the other hand, the advantages of the subsumption architecture are the following. First, its simple structure facilitates the quick and easy development of agents. Second, its simple dynamics is adequate when high responsiveness is required. Finally, it promotes a highly modular design, making the addition of new behaviours an easy task.

EXAMPLE Below, the solution to the problem example stated above is presented in the format used in the SOLUTION section.

Step 1: The environment of this problem can be described as the set of tuples (*time*, *location*, *gradient*, *object*), where:

- *time* is the instant at which the tuple refers to;
- *location* is the location of a position on the planet (usually, the whole planet is divided into zones to make the location a discrete variable);
- *gradient* is the gradient of the signal emanating from the mother spacecraft in the location; and
- *object* is the type of object occupying the location: none, sample, crumb, obstacle, base of mother spacecraft, etc.

Now, the agent only needs to perceive a small subset of the environment. Such a subset can be described as the set of tuples (*base*, *object*, *up gradient*, *down gradient*), where:

- *base* is true or false, indicating whether the vehicle is at the spacecraft base;
- *object* is the type of object in front of the vehicle (obstacle, sample, crumb or other);
- *up gradient* is the direction with the highest gradient; and
- *down gradient* is the direction with the lowest gradient.

In addition to this, we note that expressing behaviours is easier if we include the current state of the agent as another dimension in the range of the perception function (*see* function). The current state of a vehicle is formed of two variables: *carrying sample*, indicating if the vehicle is carrying at least one sample; and *direction*, representing the current direction.

Steps 2 and 3: The behaviours involved in the solution are the following, together with their perceptual inputs and tasks.

- b1. If an obstacle is detected then change direction.
- b2. If the vehicle is carrying samples and is at the base then drop samples.
- b3. If the vehicle is carrying samples and is not at the base then drop two crumbs and travel up gradient.
- b4. If a sample is detected then pick up a sample.
- b5. If crumbs are sensed then pick up one crumb and travel down gradient.
- b6. Choose a direction randomly and move to that direction.

Step 4: The inhibition relation, denoted by $\prec$, is as follows:

$$b1 \prec b2 \prec b3 \prec b4 \prec b5 \prec b6$$

The activities that can be performed by a vehicle without interacting with the environment are: change direction, and choose randomly a direction.

4.3.3 Obtaining a detailed design for the subsumption architecture

The pattern presented above can be viewed as a template in which the features of a particular application can be inserted in order to obtain the detailed design of an agent. However, as was discussed in Section 4.2.1, when the features of the particular application are provided by a high-level design, additional work is needed to elaborate the information as required by the pattern. This section contains guidelines for obtaining the information required by the subsumption architectural pattern from the high-level design obtained by a methodology.

Agents modelled through the subsumption architecture usually have either a single simple goal — decomposable into behaviours — or no goals at all since they operate on the request of

other agents. As stated above, the subsumption architectural pattern requires the following information.

- The set of environmental states that can be perceived by the agent.
- The behaviours of the agent.
- The perceptual inputs of each behaviour, which are taken from the set of environmental states mentioned above.
- The task of each behaviour.
- The precedence of the behaviours.

This information can be obtained from the high-level design as follows. First, since the set of environmental states represents how the agent perceives its environment, it is obtained from the environmental entities that the agent can read or modify, as expressed in the roles' *permissions*. These environmental entities include those needed by the agent to execute its activities and protocols.

Next, since the behaviours represent the functional aspects of the subsumption architecture, they must be associated with the roles' responsibilities. In the following we show how to map responsibilities to behaviours, but since responsibilities are formed of activities and protocols, we first state how to map activities and protocols to behaviours. First, each activity is simply mapped to a behaviour which has no perceptual inputs (so it is always triggered), and whose task is the activity itself.

Protocols, however, are more complicated. In the case of protocols involving only one message exchange, the corresponding behaviour depends on whether the agent acts as the initiator or the responder. If the agent is the initiator, the protocol can be seen as an activity whose purpose is to send the message exchanged, and the mapping for activities described above applies. On the other hand, if the agent acts as the responder of the message, the protocol gives rise to a behaviour whose perceptual input is the message and its task is the processing of such a message. In the case of a protocol consisting of more than one message, the same procedure applies, with the addition that a mechanism is needed to keep track of the order of the messages, as well as to differentiate it from other conversations. Such a mechanism can be based on the use of variables that take mutually exclusive values.

Using these matches of activities and protocols to behaviours, the way liveness responsibilities are represented in the subsumption architecture is stated below. Liveness responsibilities are formed of protocols and activities (called operands for this purpose) linked by operators of *sequence* (.), *alternative* (|), *repetition* ($*$, $+$, ^{w}), and *concurrency* (||) [134]. The basic idea for representing the liveness responsibilities is to modify the operands' behaviours, by extending in one dimension the space of perceptual inputs, to accommodate an artificial *control* variable, and appending a basic instruction to their tasks, to manipulate that variable. The exact form in

which the variable is manipulated depends on the specific expression, and its aim is to force the correct order of execution. This general idea is best explained by means of an example. To find the representation of the liveness expression $(a.b)^w$, assume that the associated behaviours of a and b are h_a and h_b, respectively, and have the following form:

$$h_a = ((v_a^1, \cdots, v_a^n), task_a),$$

$$h_b = ((v_b^1, \cdots, v_b^n), task_b),$$

where $(v_a^1, \cdots, v_a^n)$ denotes the perceptual inputs of the behaviour, and $task_a$ denotes the task of the behaviour.

The modified behaviours that represent the liveness expression are:

$$h_a = ((A, v_a^1, \cdots, v_a^n), task_a \cup \{var \leftarrow B\}),$$

$$h_b = ((B, v_b^1, \cdots, v_b^n), task_b \cup \{var \leftarrow A\}),$$

where var is the environmental entity representing the artificial variable (it is assumed that the assignment instruction $var \leftarrow a$ is executed as part of the initialisation of the agent, and that the union operation ($\cup$) denotes that the instruction is added to the other instructions of $task_i$).

The explanation is straightforward: in the beginning the value A would preclude the behaviour h_b from being triggered (since it requires a value B) and the trigger of behaviour h_a is not changed. Later, when behaviour h_a has been triggered and its task executed, the value B leaves no other possibility but to have the behaviour h_b triggered when its perceptual inputs are met.

Safety responsibilities can be represented similarly in the subsumption architecture, as follows. Since a safety responsibility is a condition that must always be true, we suggest the creation of an associated liveness responsibility, whose purpose is to re-establish it when some unexpected event causes its violation. Thus, to represent safety responsibilities we propose a twofold plan: program the behaviour tasks in such a way that the safety responsibilities hold under normal situations; and map the associated liveness expressions, as described above, to cover unexpected situations. For example, if a safety responsibility states that a vehicle must avoid obstacles, the task for changing direction must be programmed so to avoid obstacles. Additionally, a liveness responsibility must exist to correct the unexpected situation of an eventual crash.

To complete the elements required by the subsumption architecture, we show below how to obtain the inhibition relation of the behaviours. Note that the inhibition relation is used to select one behaviour when more than one matches the current environmental state. However, in the way behaviours are constructed, it is impossible for two of them to match the same environmental state if the following conditions hold.

- The values for the artificial variable are carefully chosen, so that they are unique.

- The messages are uniquely identified, for example, by means of protocol or conversation identifiers.

Thus, under this assumption of uniqueness, no environmental state matches more than one behaviour and, as a result, the inhibition relation is the empty set.

4.4 The dMARS architectural pattern

The second architectural pattern considered corresponds to the dMARS architecture, which is representative of BDI architectures. dMARS is an architecture that can be used in a broad range of applications because of its flexibility, which is achieved by means of combining several reactive and deliberative components. Associated with this combination of components, however, is the complexity of its operation, which is difficult to assimilate by novice developers. For this reason, it is important to construct tools, like the pattern presented below, that hide the general aspects of the architecture and allow the developers to focus on application specific issues.

4.4.1 The dMARS architecture

Although there are many agent architectures available, relatively few of them have been applied to solve real-world problems. One of the most notable cases is the distributed Multi-Agent Reasoning System (dMARS) [29], which is based on the Belief-Desire-Intention (BDI) model, and has been applied in arguably the most significant multi-agent applications to date.

Four data structures are commonly found in BDI agents: beliefs, goals, intentions and a plan library. Beliefs are information that the agent has about the world, which may be incomplete or incorrect. Usually, beliefs are represented symbolically, for example, as Prolog facts.

Desires or goals are the tasks allocated to the agent. In general, an agent is not expected to achieve all its desires. Those desires that an agent does choose to commit to are called intentions, and an agent tries to fulfil an intention only until it is satisfied or until it is no longer achievable. The operation of a dMARS agent is based on its plans, which are specifications of how to achieve intentions, and are stored in the plan library.

A plan consists of four components: trigger or invocation condition, context or pre-condition, maintenance condition and body. The invocation condition specifies which events trigger the plan, while the context specifies the circumstances under which the execution of the plan can start. During execution, the maintenance condition specifies the circumstances that should remain true. Finally, the body specifies the course of action needed to satisfy the plan. These actions can be sub-goals or primitive actions, which can be seen as procedure calls.

During execution, an interpreter is responsible for managing the operation of the agent, continually executing the following processes:

- observe the world and the agent's internal state in order to update a queue of events;
- generate new possible desires, by finding plans whose trigger event matches an event in the event queue;
- select one of these plans for execution (an intended means);
- if the event is a sub-goal, push the intended means onto an existing stack, otherwise push it onto a new stack; and
- select an intention stack, take the topmost plan and execute the next step of this current plan, if the step is an action, perform it, otherwise, if it is a sub-goal, post this sub-goal on the event queue.

Thus, when a plan is executed, its sub-goals are placed on the event queue. These sub-goals, in turn, trigger new plans that fulfil them, and so on. It should be noted that all the plans are generated at design time, by the agent programmer.

Specifically, there are two different modes of operation, one when the event queue is not empty, and one when the queue is empty. If the queue is not empty, an event is selected (usually the first element) and *relevant* and *applicable* plans are determined. An applicable plan is selected and its plan instance is generated. If the event is external (a newly originating event), a new intention is created and the plan instance is pushed onto it. If the event is internal (caused by a sub-goal of an existing intention), the plan instance is pushed onto the intention stack that generated that event.

On the other hand, if the event queue is empty (in which case, the operation is called intention execution operation), the first step is to select an intention. Then, from this intention the executing plan is identified, and from this plan an action or sub-goal is selected for execution. When such an action or goal succeeds, a new state is reached. If the new state is not an end state, another action is executed, otherwise the plan has succeeded.

If there are more plans in the intention, the successful plan instance is removed from the intention stack and the event that generated the completed plan is removed. If there are no more plans, the intention has succeeded and is removed, and the corresponding (external) event that generated the intention is removed, too.

Of course, there are further details that can be given of dMARS, and the interested reader should see [29] for those details. However, they are not necessary for the presentation of the pattern, and would overcomplicate and expand this chapter dramatically, so we do not present them here.

4.4.2 Pattern description

The architectural pattern corresponding to the dMARS architecture is based on the description given in [29], and is presented below.

NAME dMARS.

CONTEXT After modelling the macro level of a multi-agent system, a developer identifies that the behaviour of one of the agents can be appropriately modelled using the BDI model. The developer might not be an expert in the BDI model, so it is desirable to have a mechanism that hides general aspects and lets the developer focus on application-specific details.

PROBLEM The developer needs to specify the implementation of an identified agent. The developer has already obtained the main characteristics of the agent, mainly in terms of its behaviour. The developer is certain that the agent needs to maintain a complex mental state to achieve its goals, but at the same time it needs to yield opportune responses to fast changes in the environment.

SOLUTION In order to design an agent according to the dMARS architecture, the activities below must be performed.

1. Form the belief domain, which is the set of belief formulae representing all possible beliefs of the agent.
2. Form the set of beliefs, which comprise the information the agent has about its environment.
3. Define the events that will make the agent adopt new plans. According to the class diagram in Figure 4.4, such events are of four types: acquisition of a new belief, the removal of a belief, receipt of a message, and adoption of a new goal.
4. Define the goal domain of the agent, which is the set of all possible goals that a plan may contain.
5. Form the plan library. Plans specify how to achieve a intention and, as can be seen in the class diagram (Figure 4.4), are composed of several parts: invocation condition, context, body, and maintenance, success and failure conditions.
6. Collect the *expertise*, all the external actions that the agent is capable of performing.
7. Define functions for selecting an intention, an event and a plan. Some simple functions are provided in the *Logic* class of Figure 4.4.
8. With these elements complete the *Agent* and *Logic* classes in the class diagram of Figure 4.4.

KNOWN USES The dMARS architecture has been successfully used to build several real-world applications, for example air traffic management systems and server-side customer-service applications.

STRUCTURE The classes that form the structure of the design can be grouped into packages, as illustrated in Figure 4.3, in which packages contain the classes indicated below.

- *Agent*: the class that control of the other classes, and classes referring to events.

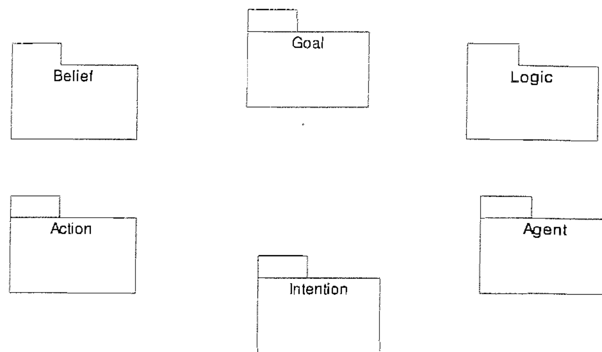

FIGURE 4.3: Packages for the dMARS architecture

- *Belief*: classes related to the representation of beliefs and triggers.
- *Goal*: classes referring to goals.
- *Intention*: classes about intentions and plans.
- *Action*: classes referring to the different types of actions.
- *Logic*: classes related to the manipulation of logical expressions and procedures for selecting plans and intentions.

The complete set of classes is depicted in the class diagram in Figure 4.4. *Agent* is the main class in the diagram, and contains the cognitive and functional elements of the agent. The *Agent* class also encompasses the control activities of the agent, determining the way in which other entities are employed. Among these entities, the *Selector* and the *Logic* classes are worth particular mention: the *Selector* class encapsulates the procedures for selecting one of several relevant plans, applicable plans, intentions and branches, while the *Logic* class contains the procedures for the logical manipulation required for other classes. Other important classes in the structure are those that encapsulate the data and functionality relating to beliefs, plans, plan instances, intentions, which are the *Belief*, *Plan*, *PlanInst* and *Intention* classes, respectively.

DYNAMICS The main operation of a dMARS agent can be divided into two scenarios: when the event buffer is empty, and when it is not empty. Figures 4.5 and 4.6 show the sequence diagram for these scenarios, respectively. As can be observed, the *Agent* class uses the services provided by the other classes to carry out the functionality of the interpreter module, which is described above.

IMPLEMENTATION It can be observed that the implementation of this pattern requires the use of procedures for the manipulation of logical formulae, for example for the unification of formulas and the composition of environments. Since these procedures are independent of the rest of the agent, they can be developed independently. Moreover, since the

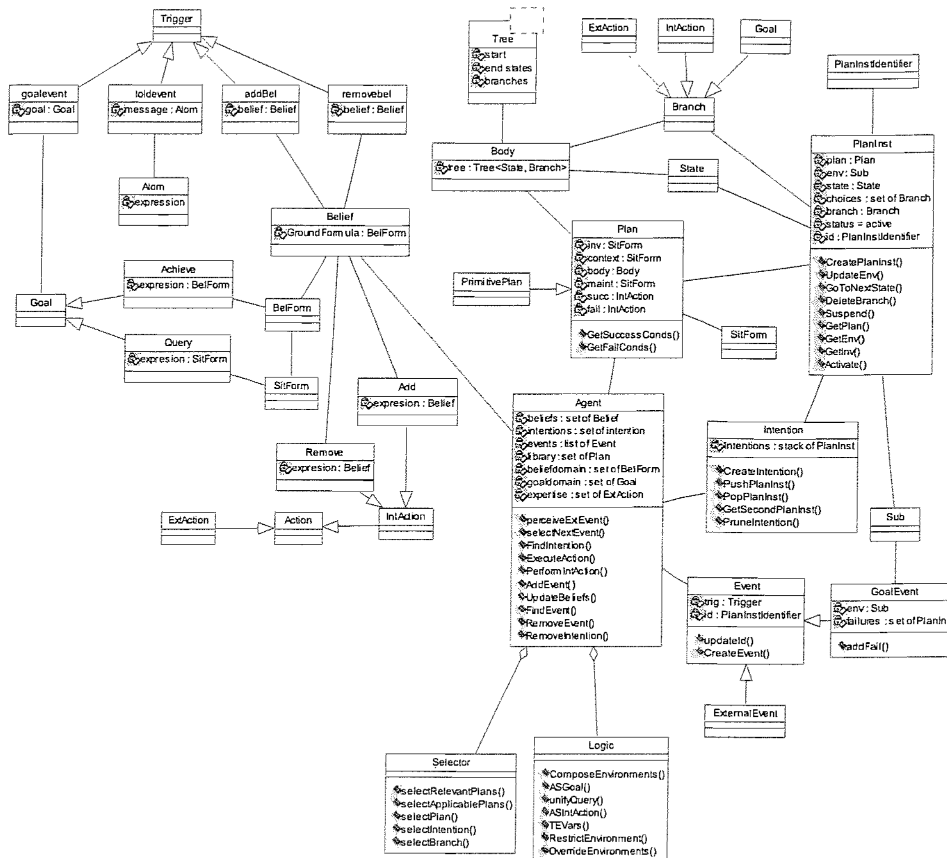

FIGURE 4.4: Class diagram for the dMARS architecture

implementation of these procedures is non-trivial and time-consuming, the use of available libraries — either commercial or free software — should be considered. In fact, the availability of appropriate libraries might be a factor when selecting an implementation platform.

EXAMPLE To illustrate the main concepts in the pattern, consider a simple example in which a robot is used for waste disposal [103]. One plan in this example may consist of *picking up and disposing waste*, and its components, according to the class diagram, are the following. The event that triggers the plan (the *inv* attribute of the *Plan* class in Figure 4.4) is that some waste appears in a particular lane, the context of the plan is that the robot is located in the same lane as the waste, and the body of the plan consists of these sequence of actions: pick up the waste, reach the bin location, and drop the waste in the bin. Reaching the bin location is a sub-plan that can be described, in turn, as a plan. Additionally,

the maintenance condition of the plan (the *maint* attribute of the the *Plan* class in Figure 4.4) is that the bin is not full, the successful action (the *succ* attribute of the *Plan* class in the same figure) is nil, and the failure action (the *fail* attribute of the *Plan* class in the mentioned figure) is that the bin is left in its original location.

CONSEQUENCES The advantages of using this pattern to implement an agent are as follows. Firstly, the BDI architecture is appropriate for modelling the behaviour of agents for a variety of domains. Secondly, the dMARS architecture is an implementation of a BDI architecture which is practical but also has a sound theoretical background. Finally, the dMARS architecture is flexible in terms of achieving a good balance between deliberative and reactive behaviours. The disadvantages of this architecture, and thus of this pattern, are that it requires significant effort from practitioners to be familiar with dMARS, due to its complexity, and that the pattern does not explicitly address the situation when the characteristics of the goals change with time.

4.4.3 Obtaining a detailed design for the dMARS architecture

BDI architectures, of which dMARS is representative, are among the most used architectures in the implementation of agent-based systems. This popularity can be explained by the facts that BDI architectures have been successfully used in real-world applications, that its flexibility suits a great variety of domains, and that there exist many implementation platforms based on BDI concepts. However, BDI architectures are difficult to assimilate because of the different concepts involved, their large number of components, and the complexity of their operation. This difficulty is even greater for non-agent specialists, as software developers usually are. It is therefore, important to provide guidelines to facilitate the process of moving from a general methodology, such as Gaia, to a detailed design. To this end, in the following we describe how to use the design models of Gaia when applying the dMARS pattern described above. That is, we consider what data is required by the pattern and how it can be determined from the information provided by the models.

We begin the description by observing, from the *SOLUTION* section of the pattern, that the information required by the pattern consists of: belief domain, beliefs, events, goals, plan library, and expertise.

This information can be obtained from the methodological design as follows. First, the belief domain can be obtained from the *permissions* of each role, since they contain the entities of the environment to which the agent have right to access or change. Second, the beliefs can be obtained from the environmental entities in the belief domain, and their corresponding values. Third, external events, which are associated with perceptions, are generated by those activities that perceive environmental entities, or process incoming messages. Fourth, the goals can be obtained from the *liveness responsibilities* of each role that the agent implements. Each alternative in a liveness responsibility (operands of the | operator) can be regarded as a specific goal,

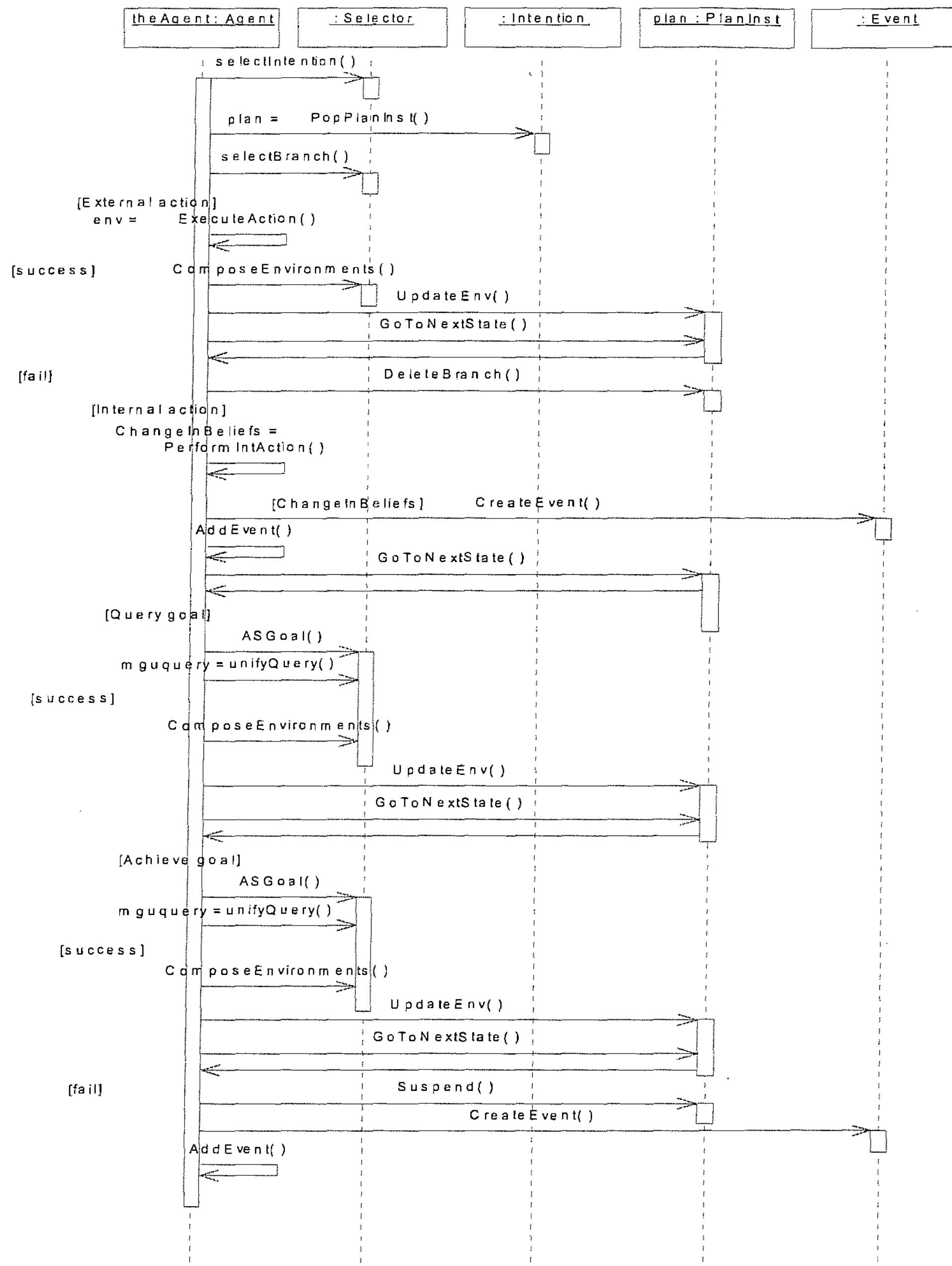

FIGURE 4.5: Sequence diagram for the case when the event buffer is empty

and the composition of the alternative — formed of protocols and services — can function as a first attempt to decompose the goal into sub-plans. In this way, the invocation of the sub-plans should be made in such a way that reflects the sequence of the protocols and services in the alternative. For the other parts of the plans, the design does not provide relevant information, so they must be determined by some other means. Finally, the expertise (the external actions that

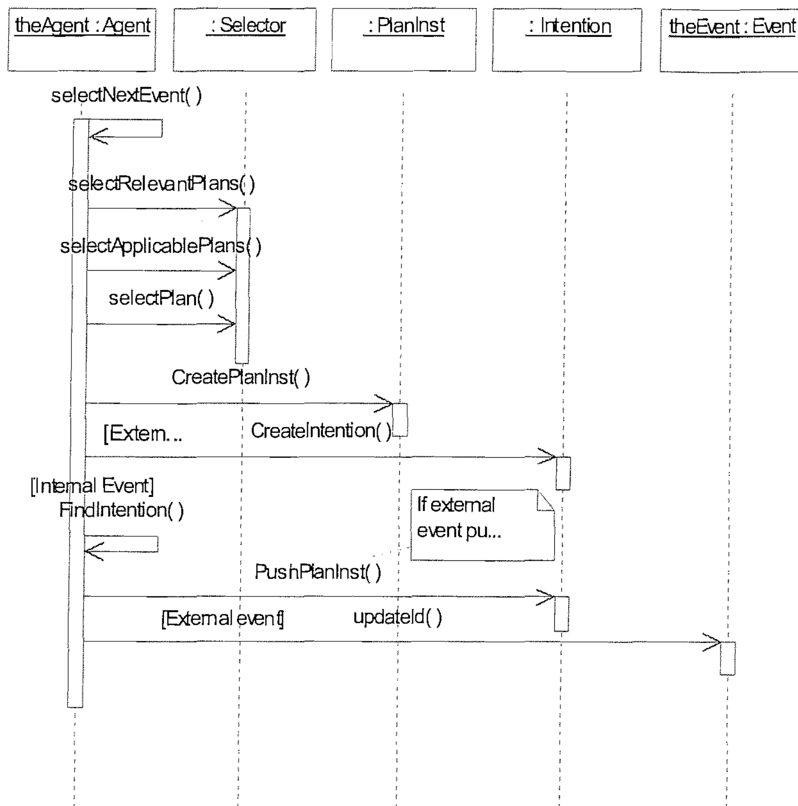

FIGURE 4.6: Sequence diagram for the case when the event buffer is non-empty

the agent is capable of performing) is obtained from those activities and services that modify the entities in the environment.

4.5 The TouringMachines architectural pattern

The last architectural pattern considered in this thesis corresponds to the TouringMachines architecture. Similarly to the subsumption and dMARS architectures, the TouringMachines architecture contains vocabulary and notions not commonly used in software engineering, making it difficult to assimilate by software developers. In this sense, the pattern presented below can act as a self-contained tool that developers can use for facilitating the construction of agents.

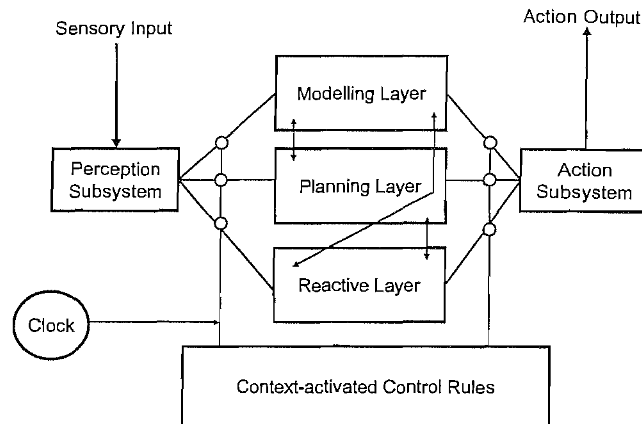

FIGURE 4.7: The TouringMachines architecture

4.5.1 The TouringMachines architecture

TouringMachines [33, 32] is a hybrid layered architecture proposed by Ferguson as part of his doctoral thesis, and is formed of reactive components as well as deliberative ones. It is layered because its core is formed of three vertically distributed layers, each with direct access to the components that sense and act on the environment. These layers operate independently and concurrently, and each generates an action as a result of every event perceived. In the following we present a description of this architecture, focusing only on the aspects needed to understand the pattern, and omitting details that are not relevant for this purpose.

The objective pursued when designing the TouringMachines architecture was threefold: to provide resource-bounded agents with the ability to be reactive; to behave in a goal-directed fashion; and to determine the impact of events — taking place in the environment — on goals (including the prediction of *what* is likely to happen in the near future). Roughly, each part of the objective corresponds to: the *Reactive* Layer, the *Planning* Layer and the *Modelling* Layer, respectively, the distribution of which is shown in Figure 4.7.

As stated above, each layer independently generates an action for each perceived event. However, an action generated by one layer may conflict with the operation of another layer, so a mechanism, or *control framework*, is needed to select an appropriate overall action. This control framework consists of two parts: a message passing mechanism between the layers (represented in Figure 4.7 as arrows between the layers), and a set of control rules that are domain specific and activated by context. By means of the control framework, one layer can alter the normal operation of another layer. For example, in a vehicle controller, the Reactive Layer can be designed to prevent the vehicle straying over the lane marks but, while overtaking another vehicle, the Planning Layer can inhibit this behaviour by avoiding the Reactive Layer from sensing the lane marks. As suggested in the figure, the architecture operates only at every click of the clock. The pattern corresponding to the TouringMachines architecture is presented below.

4.5.2 Pattern description

NAME TouringMachines.

CONTEXT A software developer has designed a multi-agent system at the macro level, which implies the identification of agents, their responsibilities, and their interactions. The next step is to model the internal structure of the identified agents. It has been determined that one of the agents must possess planning as well as reactive characteristics, so the TouringMachines architecture has been selected to model its internal structure. The developer might not be an expert in agent architectures, so it would be desirable to have a mechanism that hides the domain-independent aspects and lets the developer focus on application-specific details.

PROBLEM The problem is to design an agent whose characteristics have already been identified, using the TouringMachines architecture and an object-based design. Such a design should be understood by any software engineer who knows only basic concepts of multi-agent systems.

SOLUTION In order to utilise this architecture, the steps below must be followed.

1. Arrange the reactive behaviour of the agent (that is, situations that need a quick response) in the form of *situation-action* rules, which specify what *action* the agent must perform as a response to an *input* received by the sensors. For example, a moving agent might decide to change direction if it senses an object close ahead. This set of situation-action rules forms the core of the Reactive Layer.
2. Compile the goals of the agent into a *goal stack* in the Planning Layer.
3. Store the features of the agent's environment in a *environment database* in the Planning Layer. For example, if the agent controls a vehicle, the environmental database must contain a topological map of the world.
4. Express the plans of the agent in the form of *schemata* in the Planning Layer. A schema is a procedural structure which consists of a body, a set of preconditions, a set of applicability conditions, a set of postconditions, and a cost (in terms of computational resources).
5. For each relevant entity in the environment build one or more *models of behaviour*, consisting of a configuration vector, and the beliefs, desires and intentions ascribed to the entity. Different models of the same entity differ in the depth of information that can be represented and initial default values provided.
6. Provide lower and upper bounds within which variables of the configuration vector can vary before a conflict is declared.
7. Provide a list of *conflict resolution strategies* that the agent must follow when a conflict is detected between the observed behaviour and the planned behaviour of an entity.

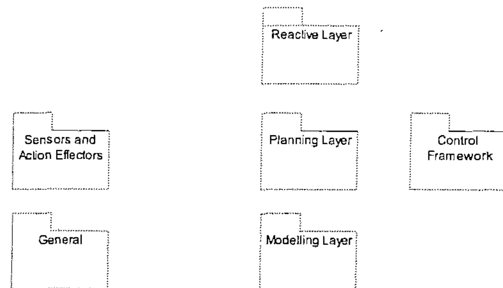

FIGURE 4.8: Packages in the TouringMachines architecture

8. Use the classes provided in the pattern to complete the design of the agent.

KNOWN USES Ferguson [33] describes an application in which the goal of a TouringMachines agent is to travel from one start point to an end point, in a simulated environment of two dimensions that is occupied by other (independent) agents, obstacles, walls and information signs. The architecture proved not only to be successful in this dynamic environment, but also flexible enough to adapt itself to different levels of uncertainty in the environment.

STRUCTURE To clarify the organisation of the pattern, the classes are grouped into packages, which are shown in Figure 4.8. The package *General* contains all those classes that are common to more than one package and usually represent basic concepts such as belief or rule. The content of the other packages can be easily deduced by their name; for example, the Reactive Layer package contains those classes necessary in the design of the Reactive Layer. In Figure 4.9 a class diagram represents the classes contained in the Planning Layer package, in which the *Planner* class contains the procedures to carry out the functionality of the layer. These procedures act on the data structures represented in the *SchemaLibrary* and the *Schema* classes, and rely on the *HierarchicalPartialPlanner* class for actually constructing the plans.

DYNAMICS The main flow of operation of the TouringMachines architecture is shown in the sequence diagram of Figure 4.10. As can be observed, the controller obtains the next event and sends it to the controller of rules, and the focus attention module of each layer. From the former, the controller receives the applicable censored rules (rules that prevent the event being fed to a layer), whereas from each of the attention modules it receives a proposed action. With this information, the controller activates the corresponding censor rule (a rule that inhibits the action of a specific layer), determines the action to execute, and orders its execution.

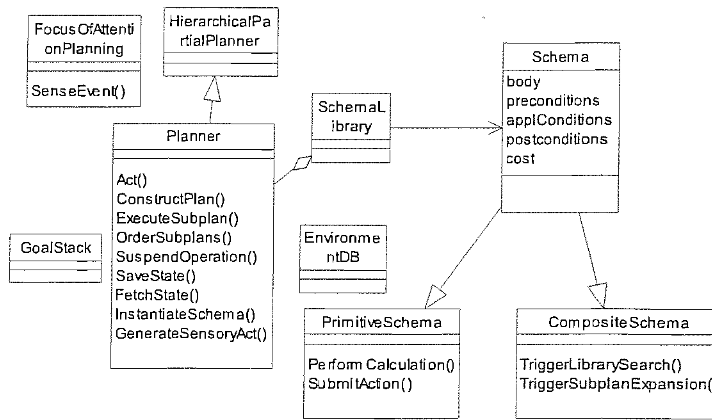

FIGURE 4.9: Class Diagram of the Planning Layer

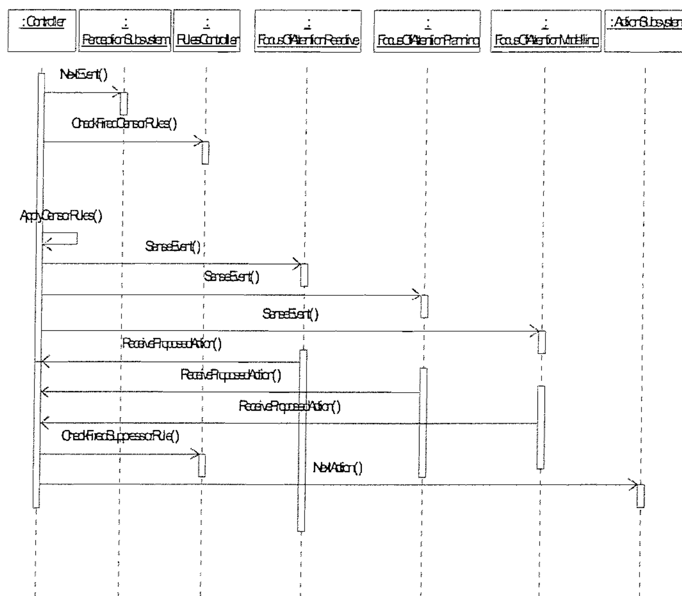

FIGURE 4.10: Main flow of operation in the TouringMachines architecture

IMPLEMENTATION Some non-trivial algorithmic components must be provided to complete the implementation of the architecture, namely a planner and a predictor. The planner required is a hierarchical partial planner that can interleave plan formation and execution, and defer committing to specific sub-plan execution methods or temporal orderings of sub-plans until absolutely necessary. It must allow the possibility of being regularly pre-empted and its state being suspended for subsequent use, and must use a combined earliest-first depth-first search for constructing the plans. The predictor must be capable of predicting the behaviour of an entity by making a temporal projection of its characteristics (configuration vector) in the context of the current world situation and the entity's ascribed intentions.

CONSEQUENCES The advantages of using this pattern to implement an agent are the following. First, the architecture provides the agent with both reactive and deliberative features, whose balance can be tuned to a specific application. Second, by modelling the behaviour of the agent itself and of other entities, the architecture can accurately predict potential conflicts in their goals, making it possible for the agent to change its behaviour to avoid them. The disadvantages of this architecture are the following. First, before using the agent in a real situation some tuning must be performed, which basically means finding, by trial and error, the values of the parameters that make the agent suitable for a particular application. Second, the architecture involves the use of concepts that a novice practitioner may find complicated, such as conflict resolution strategies.

EXAMPLE In the application described by Ferguson [33] about agents travelling from one point to another, the Reactive Layer contains situation-action rules for avoiding collisions between agents travelling along the same lane. Such rules are simple to state; for example, for an agent to avoid collision with the object in front, the *situation* part of the rule consists of checking that the object is in front, that its velocity is less than the agent's and that the distance between the object and the agent is less than a threshold. Additionally, the *action* part of the rule commands the agent to reduce its velocity.

In the Planning Layer, determining a route is a goal of the agent, and is thus stored in the goal stack. In relation to this goal, there must be a plan schema to plan a route, whose elements consist of: no preconditions; a body consisting of three tasks, get a route, get the speed of a route, and follow the route; an applicability condition that states that these tasks must be performed in that order; no post-conditions; and a cost of three units (presumably one unit per task).

4.5.3 Obtaining a detailed design for the TouringMachines architecture

The TouringMachines architecture consists of reactive, planning and predictive components, making it suitable — at least in principle — to design a broad range of agents. However, the information required to use this architecture, and consequently its pattern, is not directly obtained

from a high-level design. Thus, an additional process is necessary in order to obtain an agent design by using the TouringMachines pattern. Such a process is described below.

Summarising, the elements required by the TouringMachines architecture are the following.

- Environmental states (in the perception subsystem). States of the environment that the agent can recognise.
- Censor rules (in the control framework). They are if-then type rules that check the presence of particular sensory objects and prevent them from being fed to selected layers.
- Suppressor rules (in the control framework). These are if-then type rules that check the presence of particular actions and inner states and prevent them from being fed to the effector subsystem.
- Situation-action rules (in the Reactive Layer). These rules form the core of the Reactive Layer, associating environmental states to actions, so that the action is (potentially) executed when the corresponding environmental states hold.
- Focussing rules (in the Planning Layer). These rules filter the entities that are considered relevant for planning tasks. They are built on pre-defined predicates which are mainly domain-dependent.
- Schemata (in the Planning Layer). A schema is a high-level description of an achievable task, and is used by the planner to build plans to pursue tasks. It consists of: *body*, the actual instructions that carry out the task; *preconditions*, states of the world that must hold for the task to be performed; *applicability conditions*, that specify the order in which the body steps can be performed; *postconditions*, states of the world that must hold for the schema to complete; and *cost*, the amount of resources consumed by the execution of the schema.
- Database of the world (in the Planning Layer). A database is used to store knowledge about the environment of the agent; for example, for a navigating agent the database contains a topographical map of its surrounding world.
- Focussing rules (in the Modelling Layer). These are similar to those of the Planning Layer, but with the specific level of abstraction required by the Modelling Layer.
- Models (in the Modelling Layer). Models are representations of the entities external to the agent such as other agents and environmental entities. Models consist of four parts: a *configuration* for expressing the characteristics of the entity, such as a unique identification and some other domain-specific features; and the beliefs, desires and intentions ascribed to the entity.
- Conflict library (in the Modelling Layer). The conflict library specifies the possible goal conflicts and the way in which they are resolved, and is formed of conflict resolution rules,

each consisting of three parts: the conflict identifier to which they refer, the goal that is under conflict, and the procedure to follow for an eventual recovery.

In order to map a Gaia-type design to this architecture, we suggest beginning by defining the environmental states, which includes the identification of the information perceived by the agent. This is done in a similar way to other architectures, noting that each entity observed must be uniquely identified. First, since the set of environmental states represents how the agent perceives its environment, it must be obtained from the the environmental entities that the agent can read or modify, as expressed in the roles' *permissions*. These environmental entities must include those needed by the agent to execute its activities and protocols, as well as the messages exchanged with other agents during the execution of protocols.

Considering that the Reactive Layer contains those situations that require a quick reaction, the candidates to be modelled in this layer are the following.

- Safety responsibilities with simple recovery procedures. The reason for this is that safety requirements are usually based on the conditions of perceived information, and include a recovery action; when violated they require a quick recovery. For example, in a pipeline-type production organisation, a safety responsibility might state that the processing flow of items must be maintained to a constant. If an expected situation causes a reduction of the flow, the appropriate action must be triggered, for instance to increase the speed of a motor. In a TouringMachines this safety requirement can be naturally mapped to a situation-action rule that monitors the difference between the constant flow and the agent's flow, and the action increases the speed of the motor.

- Activities that consist of receiving a message, processing its content in a simple way, and replying to the message. These are common activities for some type of agents, for example those acting as *wrappers* of legacy software. The key point here is to decide if the process of the content is actually simple enough to be completed in a short time.

- Any other actions triggered by external events that are simple and require quick response. These requirements obey the fact that the actions will be part of the reactive behaviour of the agent.

The rest of the agent functionality must be represented in the Planning Layer. Thus, the designer must provide a decomposition of the corresponding safety and liveness responsibilities into sub-plans by means of the schemata referred to above. It must be noted that more than one sub-plan can be available for the same task, in which case the sub-plan to use is selected by cost. However, doing this requires the use of planning techniques that belong to the field of artificial intelligence rather than agent computing, and it is out of the scope of this thesis to describe a form of plan generation for responsibilities.

We suggest using the database of the world for storing the static information about the world that is normally stored in the *beliefs* component in other architectures.

However, a design produced by Gaia does not provide enough information for obtaining the other elements required by the architecture, so we can only mention some general guides. Focusing rules are straightforward to define, although highly domain dependent. The key point when defining the focusing rules is to note that they are used to filter all the possible perceptual inputs and propagate only the relevant information at the correct level of abstraction.

In the case of the modelling library, which consists of one model for each agent in the system and each entity in the environment, each of the models must be constructed. The first part of the model consists of a characterisation of the entity or agent, including its unique identifier. The rest of the model applies only to agents and specifies their beliefs, desires and intentions. These models are used to predict the behaviour of the entities and thus foresee and solve possible conflicts. However, it must be noted that accurate predictions in open systems is, in general, not possible, since the exact behaviour of some agents in the system is unknown at design time.

Finally, the conflict library contains the procedures to solve a conflict that arises when pursuing two or more goals.

Based on the above analysis, we can conclude that from a Gaia-type design it is difficult to obtain a detailed internal structure for architectures such as TouringMachines, since the designer must carry out complex activities such as a plan decomposition for each goal, and the explicit representation of beliefs, desires and goals for each of the agents of the system. Also, the TouringMachines architecture might not be appropriate for open systems, since the agent must have an accurate representation of every other agent's beliefs, desires and intentions.

4.6 Towards a general pattern

The previous subsections show the architectural patterns corresponding to representative architectures. We envisage three approaches for extending these results to a broader range of architectures (note that these approaches are not mutually exclusive). The first approach is to populate the catalogue of architectural patterns so that more agent architectures are considered. Populating the catalogue can be achieved by making the catalogue available to a community of developers, since this would speed its population and, through active feedback, the quality and accuracy of each pattern would be increased. As the number of patterns grows, some upgrades to the catalogue would be necessary, for example to consider a classification of the patterns by domain of application, and the inclusion of facilities to assist in the selection of the appropriate architecture for a specific application. Furthermore, the patterns may evolve to include code for specific platforms. However, this is a long-term approach, due the time it takes for a natural growth a significant number of patterns, receive feedback and update the patterns.

A second approach towards generalising the results of the previous subsections would be by means of a *general* pattern. Such a pattern would encompass the functionality of a range of architectures, allowing practitioners to specialise it for a specific application by choosing the

appropriate components and interconnections. The key disadvantage of this approach is the highly specialised background a practitioner must possess to be able to select the elements that are relevant for their purposes, assemble them, and finally generate the required pattern.

Finally, the third approach consists of guidelines to assist the development of a pattern for any other architecture. The disadvantage here is that, since each architecture is different, some parts can only be sketched, and there is always the risk that some aspects are not applicable to another particular architecture. However, in this thesis, we adopt this approach because it can be used to populate the catalogue (i.e. the first of the three approaches), and is more in accordance, than the second approach, with our aim of providing tools that can be used by typical software developers.

4.6.1 Guiding the development of an architectural pattern

There exist a large variety of agent architectures. This variety is beneficial because no single architecture provides adequate solution for all types of applications. Since it is not the purpose of the catalogue to reduce this variety, we need to find ways to allow the incorporation of other architectures into the catalogue, and is described below. Our solution takes the form of methodological guidelines for developing a pattern not considered in the catalogue. This, however, is not a detailed procedure. In fact, due to the huge diversity of characteristics exhibited by agent architectures, we believe that is not viable to construct a procedure that would be, at the same time, practical and more specific.

1. It seems obvious that the first step is to find the documentation of the agent architecture in question, but actually not all documentation is suitable for the purpose of developing guidelines for the architecture for several reasons: some documents focus on *how* the agent *behaves* instead of *how* it *achieves* that behaviour; some documents describe an architecture only for a specific problem; and some documents are vague in their description of the architecture. When looking for appropriate documentation, we need to focus on those that include a procedural viewpoint of the architecture. Documents that formalise the operation of the architecture are excellent for this purpose, but they are scarce.

2. All the essential concepts of the architecture must be well understood, observing that some terms have different meaning for different authors and even in different documents by the same author. We must complete the description of those terms that are just poorly described in the documentation.

3. A good form of starting the construction of the pattern is by defining the structure of the architecture in a class diagram. Except for the simplest architectures, it is convenient to divide the whole structure into parts by using *packages* (a graphical notation to group elements such as classes and relationships). Regarding the question of which packages to

define, we can say that it depends mainly on the size of the architecture and its composition, since some architectures are highly modular while others tend to be monolithic. In the case of those architectures that possess an intrinsic modularisation (e.g. layered architectures), it is sensible to have a package for each of the modules. At least one package should be reserved for input/output components such as sensors, actuators and message passing mechanisms.

4. All architectures include components to interact with their environment, which are are mainly of two types: components to sense and act on the environment (sensors and effectors, respectively), and components to send to and receive messages from other agents (message passing mechanisms). Without loss of generality, sensors and effectors can be treated as information entities because even physical devices can be controlled by means of computational interfaces. We deal with sensors and effectors below, while message passing mechanisms are considered subsequently.

5. Sensors and effectors are simple in concept and structure and can be easily separated from the rest of an agent. For this reason, it is easy to devise a general interface to express the functional characteristics of these components, leaving only its implementation specific to the platform. Effectors are the simpler of these two types of components since their operation is generally under the command of the agent. Because of this, an effector can be appropriately modelled as a class — shown in Figure 4.11 — whose main operation is receiving an action as an input parameter, and delegating to another (interface) class the performance of the action. If the rate of requests exceeds the rate of processing the requests, a buffer might be used to store the excess. Subsequently, the actions in the buffer might be processed on demand or at the discretion of the effector. This basic operation is depicted in Figure 4.12 by means of a sequence diagram.

   On the other hand, sensors come in two flavours: active and passive. These are depicted as specialisation classes in the diagram of Figure 4.13. Passive sensors act on request, while active sensors perceive the environment as soon as a relevant event occurs. Implementing passive sensors is simple and it is enough to have a class whose main operation returns the most recent event or the next one in a buffer. Regarding the type of value returned by this operation, it is advisable to define a type general enough to fit all the types of events that may occur (class *EnvironmentalState* in the figure). By contrast, active sensors need to announce that new information is available. This can be achieved by having an operation — *NewSensedInfoAvailable* in the figure — that indicates whether or not a new chunk of information is available or by knowing beforehand the operation to be called. In relation to the implementation of active sensors, it is also important to establish what to do with unread information when new information is obtained, the most common strategies being to discard them, or to store them in a first-in first-out buffer. This is indicated in the figure by means of the operation *SelectMode*.

6. Message passing mechanisms are used by an agent to communicate with other agents in the system. From the viewpoint of functionality, it is sufficient to have a class with

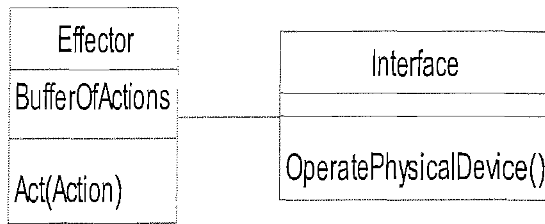

FIGURE 4.11: Modelling the structure of effectors

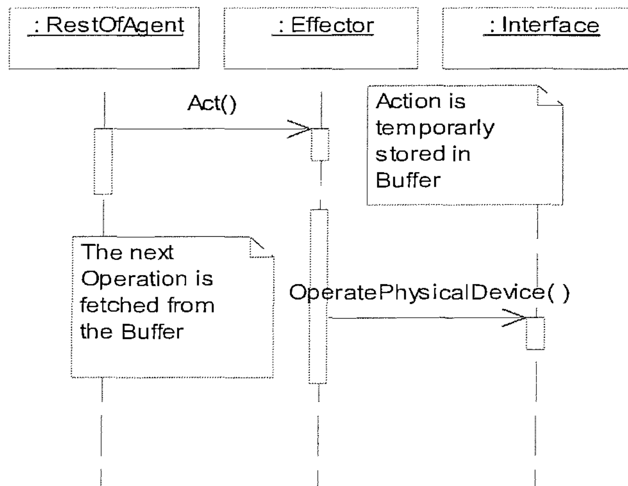

FIGURE 4.12: Modelling the dynamics of effectors

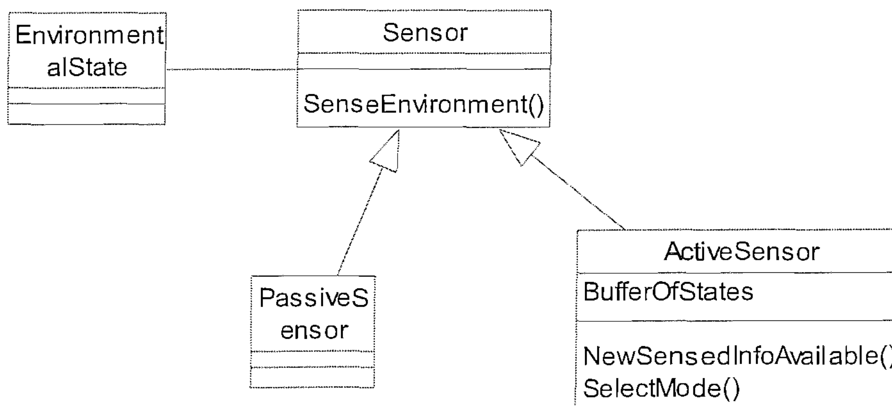

FIGURE 4.13: Modelling sensors

two operations, one to receive and one to send messages. Such a class may also include operations to parse and synthesise the language in which the messages are expressed. However, some other aspects of message passing are highly platform dependent, such as the form in which to denote the parameters of the communication, such as the sender, the receiver and a time-out period.

7. The existence of a reactive component is common in several architectures because it provides an effective way to deal with situations that require opportune responses to changing conditions of the environment. A general means of modelling a reactive component is to view it as a *controller* of *behaviours*. Behaviours consist of two parts, one part specifying the environmental conditions that fire the rule, and another part to specify the actions to be performed when the behaviour is fired. The controller is in charge of checking which behaviours are fired and selecting one or more of the corresponding actions for execution. It has been suggested that adding a state as a third component in behaviours can be convenient in several situations. On the other hand, if the architecture possesses other components (such as deliberative or planning components), the overall components can be arranged in different forms.

8. Apart from a reactive component, many architectures include at least one non-reactive component which may perform one or several tasks such as planning, predicting, scheduling, and coordination. When one or more of these components are present, they can be arranged in different ways, for example in *layers*, both horizontal or vertical. In a horizontal distribution, each of the components, or layers, has access to the sensed information as well as to the effectors, whereas in vertical layering only one layer has access to the sensed information and only one layer has access to the effectory capabilities (the same layer in some architectures). The specific number of layers and the functionality of each layer vary for each architecture. For example, the TouringMachines architecture consists of three layers, the lower layer being a reactive component, the middle layer containing reactive plans to be used according to the situation, and the upper layer being a planner which constructs plans and finds the best action to perform in order to achieve a goal.

9. Whichever components are present in an agent, it is important that they are distributed in modules, that is, separated by function with their interactions clearly identified.

10. BDI architectures are some of the most used architectures for developing agent-based applications. However, it must be noted that although all BDI architectures are based on the same basic concepts, they vary widely in terms of composition and functionality

We conclude that, with the purpose of designing a general architectural pattern, only some aspects of an agent can be generalised; for example, the input/output interfaces and the reactive component. However, due to the great variety of architectures, not much can be said about other components such as planners, coordinators, and predictors since they are not present in all architectures and, even when they are present, they vary considerably in composition and in the role they play in the agent functionality.

4.7 Related work

Some current methodologies use an explicit representation of the internal structure of the agents by means of agent architectures. Kearney et al. provide a general architecture as part of the MESSAGE [78] methodology, in which an agent is formed of at most four layers: the Perception and Communication Layer (PCL), the Decision and Management Layer (DML), the Domain Layer (DM), and the Resource Layer (RL). The PCL gets information from the environment and interacts with other agents. The DML controls actions of the agent through deliberative decisions, the DL groups domain specific entities, and finally, the RL includes the internal resources the agent may need. Kearney and colleagues state that such an architecture may be configured to satisfy specific applications. For example, reactive agents would use only the Perception and Communication Layer and the Resource Layer, whereas deliberative agents would employ all the layers. However, no further details are provided and, after all, the decomposition of any agent into only four layers is a coarse analysis. For instance, decomposing the structure of a reactive agent into only two layers (the PCL and the RL layers) does not provide a comprehensive characterisation.

INGENIAS [96] is a methodology that evolved from MESSAGE. It emphasises the construction of models for each relevant part of the system, in particular agents. The models are designed according to *meta-models*, which are descriptions of the entities, their relationships and the constraints allowed during model construction. In this way, for example, any agent is assumed to include modules such as mental entity, mental state, a set of roles to be played, and a set of tasks. Although this approach offers a rich internal representation of agents, we believe that it may lead to designs which do not have any theoretical background. This gap between design and theory makes it difficult to establish the conditions under which the agents will operate correctly. Also, this approach leaves the developer alone with the burden of *adjusting* the architecture to specific applications. In our work, we provide the developer with concrete designs of well-known architectures.

MaSE [85] is another example of a methodology that employs agent architectures to model the internal structure of an agent. During the *Assembling Agents* sub-phase, the architecture and its modules are defined by the developer, but both can be selected among those pre-defined by the methodology. Pre-defined agent architectures (called architectural styles) include reactive, knowledge-based, planning and BDI architectures. The architectural styles and the components are described using a language similar to UML, considering only static aspects and leaving out the description of control. In our work, we include a description of the dynamics of the components of the structure, and our goal is to obtain a general method for any architecture.

Tropos [81] also uses an agent architecture to represent the internal structure of agents, but it is tied to a BDI architecture. During the detailed design phase, the abstractions used in the previous phases (e.g. example actor, goal and task) are mapped to BDI concepts (agent, belief, desire and intention). Later, during the implementation phase, these BDI concepts are mapped onto

constructs provided by the ZEUS platform. One drawback of this approach is that it depends on a specific architecture, as well as on a platform.

In contrast to these methodologies, an alternative approach is to obtain an implementation directly from the design specification, without an intermediate explicit representation of the internal structure of the agents. This is the case in [94], where Massonet et al. present a case study to show the transition from a design specification obtained with MESSAGE to an implementation using the JADE platform. However, we have identified two limitations in this approach. First, since no explicit agent architecture is used, it is difficult to envisage for which type of applications the approach is suitable, and how it compares to other approaches. Second, this approach clearly depends on the methodology and platform used, which makes it unsuitable for *open* applications.

4.8 Conclusions

The adoption of the agent-oriented approach in industrial environments largely depends on the existence of comprehensive software tools that assist developers in key parts of systems design, for example during the design of the internal structure of the agents. Although it might be argued that agent internal development can be carried out by means of traditional software engineering — for example, object-oriented methodologies — these techniques present some drawbacks when used on their own. To be more specific, consider the Unified Software Development Process [69] as representative of object-oriented methodologies, in which the concepts of pro-activness, autonomy and goals do not have direct representation. Pro-activness and goals have no direct representation because objects are not capable of pursuing goals. Similarly, autonomy has no direct representation because objects are not capable of select their own course of action. Thus, to be used effectively for agent development, object-oriented methodologies need to be complemented with other mechanisms.

Agent architectures provide a powerful mechanism on which tools for modelling the internal structure of agents may be built, but they are difficult to use *per se*, because their descriptions are not targeted at software engineers.

In this chapter we have presented a framework in which agent architectures are used to build representations of the internal structure of agents, described by means of design patterns. We have also included three patterns describing some of the most cited agent architectures.

A catalogue containing patterns like these has multiple benefits. First, these patterns allow agent architectures to be viewed as tools to model the internal structure of agents. Second, a non-agent expert can use the catalogue to learn about agent architectures. Third, the catalogue facilitates the comparison of different architectures and the selection of the most appropriate one for a particular application. Fourth, the use of patterns speeds up the development process. Finally,

by reusing general solutions, the use of patterns allows us to concentrate efforts on domain-specific aspects.

Chapter 5

An incremental and iterative methodological process

In Section 2.5 we highlighted the importance of software methodologies for the development of systems in a systematic and controlled form. We also noted that methodologies consist of models, tools, and a process that is arguably the core of a methodology, describing the set of activities needed to carry out the development of a system. Thus, the process details *which* activities to perform, *how* to perform them, and in *what order*. At present, there exist a considerable number of agent-based methodologies, and thus of development processes. The types of activities considered in current processes are rich and varied, and so are the techniques to perform them. However, the order in which the activities are performed has been explored much less, and is usually in a sequential form. Although simple in concept, the problem with this sequential form is that it is not suitable for tackling complex and large systems because of the difficulty of designing a complete system in just one single iteration. Consequently, without the use of a more powerful approach for decomposing the development process, such methodologies have serious drawbacks in terms of the type and size of applications to which they can be applied.

The *incremental iterative approach* is one such approach that has been successfully used in object-based software engineering to divide the development process into more manageable units. Basically, this approach consists of decomposing the whole development of a system into several mini-projects [69], in which each mini-project follows the traditional flow of requirements, analysis, design, implementation and testing, and results in executable code that produces *increments* in the functionality of the system. The development of each mini-project starts from the specification obtained during the previous mini-project and is seen as an *iteration* that continues the workflow of development of the complete system. In order to be useful, the decomposition of the system into mini-projects must be carefully planned, usually with the aims of extending the functionality of the product and reducing the risks of failure of the whole project.

The application of the incremental iterative approach to practical problems provides multiple benefits in terms of work organisation, since the parallelism of the development activities is increased, as well as user feedback. A good level of parallelism helps to keep the project under time and budget constraints, and adequate user feedback helps to develop the *right* system, since the product of each iteration can be presented to the user, and the corrections included in the next release. However, in spite of its many benefits, the incremental iterative approach has not been applied in most agent-oriented software methodologies. In contrast, most agent methodologies exhibit just a linear sequence of steps, or the existence of iterations is only vaguely outlined.

In this chapter we present an incremental iterative process based on the Gaia methodology [134]. To this end, we first present the general characteristics of the process in Section 5.1. Then, and in accordance with how the process is decomposed, we describe the workflows — requirements, analysis, organisational design, agent design, and implementation — in sections Section 5.2, Section 5.3, Section 5.4, Section 5.5 and Section 5.6, respectively, and iterations in Section 5.7. Finally, in Section 5.8 we analyse related work and present our conclusions.

Where appropriate, the workflow descriptions have been divided into two parts. The first part presents the *artefacts* — as graphical models are named in the Rational Unified Process (RUP) [117] — and the second part presents the activities carried out during the workflow, which typically involve one or more of the artefacts. With the aim of providing the reader with a general view of the process, in Figure 5.1 we have depicted the workflows and their artefacts. If read from top to bottom, the figure shows the workflows in the order they are realised in the process. In the area corresponding to each workflow, the artefacts are represented by rectangles. Note that, for reasons of presentation, in the organisational design workflow we have also included the *organisational structure*, although it is not strictly an artefact. Also, the requirements analysis and the implementation workflows do not contain artefacts since they are not considered in depth in this thesis. In particular, requirements engineering is not considered in detail in this thesis, because its complete analysis lies beyond the limits of agent-based computing, involving areas such as goal-oriented software engineering [22] and aspect-oriented software development [49] (requirements engineering for agent-based systems is not tied to the agent approach, but can be accomplished through other approaches). Such an analysis would significantly increase the length of the thesis but would not contribute to improve the state of the art of agent-oriented software engineering. Additionally, implementation is not considered in detail in this thesis because it largely depends on specific tools (programming languages, toolkits and execution platforms). Although in any development approach the implementation process depends on the tools used, in the agent approach this is exacerbated due to the heterogeneity of the tools and the lack of standards.

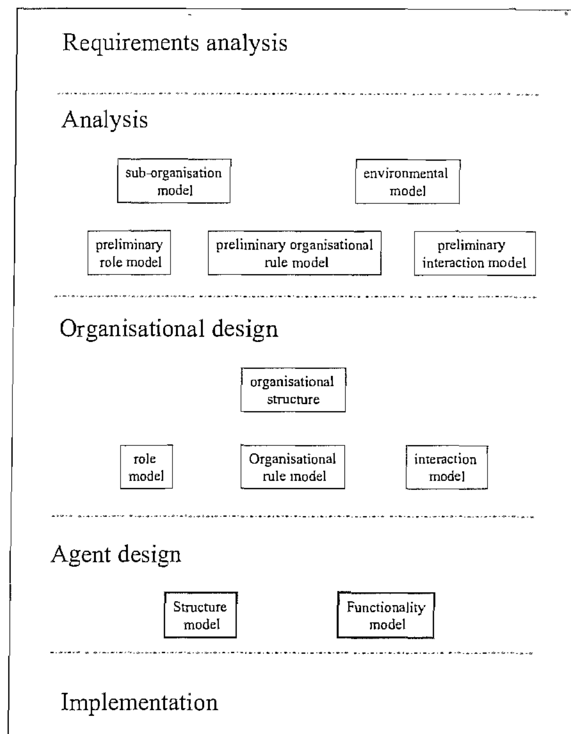

FIGURE 5.1: The workflows in the process and their artefacts

5.1 The approach

The main goal of applying the *incremental iterative approach* to the development of a system is to reduce the risk of producing the *wrong* system, and the risk of exceeding delivery times. In order to do so, the approach decomposes the development activities in two dimensions. The first dimension is similar to traditional ways of developing software, decomposing the development into requirements analysis, analysis, design and implementation (in this chapter we refer to each of these parts as *workflows*). The second dimension in which the development is decomposed is by means of *iterations*. Each iteration consists of the application, to some degree, of all the workflows mentioned above, with several iterations during the whole development cycle. Early iterations focus on the first workflows, requirements and analysis, while subsequent iterations focus on design and implementation, thus delivering executable versions of the system. Here, each new executable delivered extends the functionality of the previous one. These executable deliveries are usually internal, and are useful for evaluating the *correctness* of the system, as well as for obtaining user feedback. Also, these executables can be used as a tangible measure of the progress of system development. This decomposition of system development is depicted in Figure 5.2, in which the workflows and iterations occupy the Y-axis and X-axis, respectively. In this figure, the amount of effort dedicated for each phase and workflow is indicated by the area of the geometric form; for example, the base of the triangle in the first iteration is placed around the requirements workflow, which means that most of the effort in the first iteration is dedicated to the requirements.

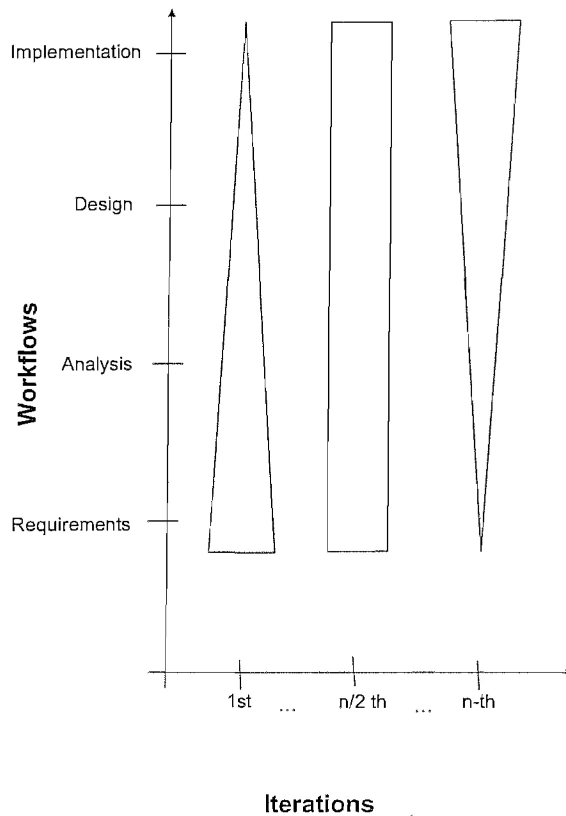

FIGURE 5.2: Workflows and iterations of the process

The sections below describe each of the workflows and a generic iteration, in the natural order they appear in the process, as depicted in Figure 5.3. Note that the design has been divided into organisational design and agent design, reflecting that each addresses very different aspects of a system. In the same figure, the existence of iterations has been indicated by an arrow going back from the last workflow (implementation) into the first workflow (requirements analysis). Most workflows are based on the Gaia methodology process, but contain some aspects that are not included (or not explicitly included) in Gaia, such as techniques, activities, and the use of the workflows in the context of the incremental iterative approach. In particular, in Figure 5.1, the models not considered in Gaia (the structure model and the functionality model) have been highlighted by means of thick lines. The description of the workflows include the artefacts realised during the workflow, and the activities required to realised them.

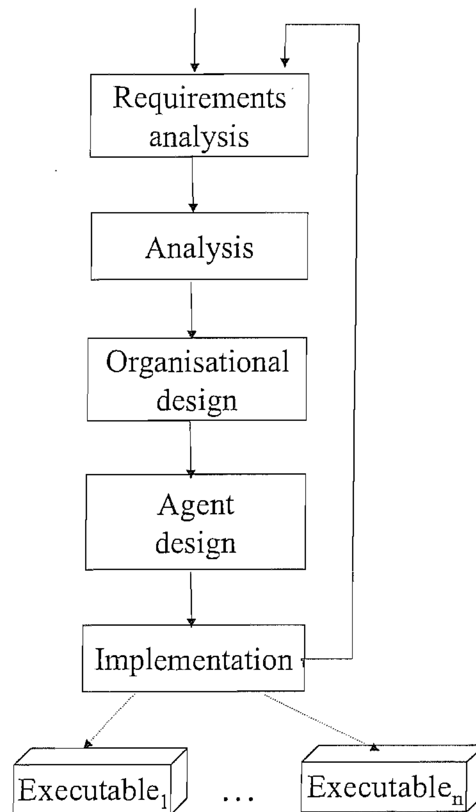

FIGURE 5.3: Workflows of the process

5.2 Requirements analysis

Although the requirements workflow is not considered in detail in this thesis, we include it here because we need to state what is expected from it; that is, what information is to be provided to the analysis workflow by the requirements workflow.

The role of the requirements phase is to gather and organise the information about the capability that the system must possess and present it in a document from which the analysis of the system starts. The exact form of achieving this goal depends on the specific approach used, but is not considered in this thesis, because its complete analysis lies beyond agent-based computing, as can be exemplified by goal-oriented requirements engineering [22]. This does not mean that the agent approach cannot be applied to this workflow, since some work does exist in that direction, for example agent-oriented requirements engineering [130].

What is expected from the requirements phase is a document that contains the following:

- An overall description of the system, and of the vocabulary needed to understand this description.
- A description of the environment of the system, the boundaries of the system and its environment.
- A description of the functionality of the system, particularly its goals. If possible, this should include the priorities of the different goals and sub-goals.
- Insights about non-functional requirements, such as the expected (or hoped for) number of users.
- The authority that the main actors in the system have over each other.
- The rules and constraints that restrict the operation of the system.

5.3 Analysis

The analysis workflow copes with understanding the system and its goals, sub-systems, elements, rules of behaviour, and the environment that surrounds it, but does not provide solutions for how to achieve the desired functionality of the system. In this way, in the analysis, understanding the functionality of the system is more important that providing a rigorous specification of the system. This is reflected in the way the models are expressed, using a language that is closer to the vocabulary of the application domain than to a formal language.

The analysis takes as input the information provided by the requirements analysis. By means of artefacts, this information is structured and organised in such a way that it can be better understood, consulted and modified. In the following subsection we describe these artefacts.

5.3.1 Artefacts

The artefacts in the analysis phase are arguably the most important artefacts of all the process, since they are used, in one way or another, in the other workflows. These artefacts are: the sub-organisation model, the environmental model, the preliminary role model, the preliminary interaction model, and the preliminary organisational rule model. (The last three of these artefacts were discussed in Section 3.2.1, although the organisational rule model was not explicitly referred to as *preliminary* then).

- The sub-organisation model decomposes the whole system into separated, and usually weakly-connected, sub-organisations, and can reflect the physical distribution of the components of the system, or can be based on other criteria, such as differentiated functionality, the existence of legacy software, or simply modularisation to facilitate the design.

- The environmental model consists of a list of the entities in the environment, together with their elements and the rights the different roles have to access them.
- The preliminary role model is formed by the roles in the system. Each role represents an entity that is in charge of accomplishing one or more responsibilities by using environmental entities, carrying out activities and interacting with other roles through protocols. Roles are described by means of their responsibilities and permissions to use the environmental entities.
- The preliminary interaction model consists of all the interaction descriptions of the system. Such interaction descriptions, or protocols, are formed from the roles that participate in the interaction and the data involved.
- Finally, the preliminary organisational rule model encompasses all the organisational rules of the system. During the analysis, the organisational rules are identified and described informally in plain text. Later, during the design, when all the elements of the system have been sufficiently defined, the organisational rules are expressed in a formal language.

5.3.2 Activities

The activities involved in the analysis workflow, and the sequence in which they are performed, are shown in Figure 5.4. As can be observed, first, the limits of the system are determined, the sub-organisations of the system are identified and the sub-organisation model is generated. Second, the entities in the environment are identified and described. Next, the main roles of the system are identified, together with their main responsibilities, and the preliminary role model is partially specified. Then, the interactions between the roles needed to accomplish their responsibilities are identified, and used to create a preliminary interaction model. Finally, the organisational rules are identified and described informally as plain text.

5.4 Organisational design

The organisational design proceeds the analysis, thus most of the organisational design artefacts are extensions or refinements of those of the analysis. Furthermore, the design artefacts are targeted at developers rather than at users, and so are described in a more precise and technical form. However, the scope of the design does not include the actual way in which the system is codified, since that is the concern of the implementation workflow.

While in other development approaches the design is regarded as a one-part workflow, in multi-agent systems it is convenient to divide the design into two parts because this reflects the intrinsic nature of multi-agent systems. The first part, the *organisational design*, is concerned with modelling the *interaction* between agents, whereas the second part, the *agent design*, is concerned with modelling the *internal structure* of the agents. In this section we describe the former, while

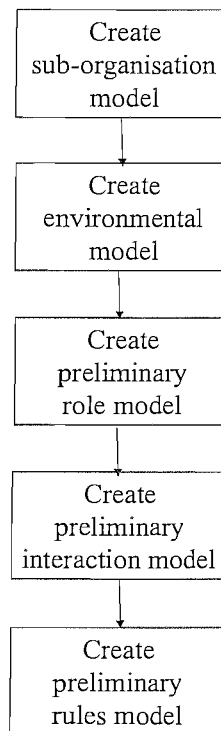

FIGURE 5.4: Activities of the analysis

the latter is described in the following section. There are two main tasks during organisational design, one of which is to find an appropriate organisational structure to model the system, and the other is to refine the artefacts obtained in the analysis. The artefacts and activities involved in pursuing these tasks are described in the following subsections.

5.4.1 Artefacts

There are four artefacts in the organisational design: the organisational structure, the role model, the interaction model, and the organisational rule model. To avoid confusion with the corresponding analysis models, sometimes we refer to these three last models as the *definitive* models. As was discussed in Chapter 3, an organisational structure serves as a framework in which agent interactions are modelled, and at the same time provides a structure that supports the development of the system, just like *software architectures* are employed in the Rational Unified Process [117].

Because of their importance, organisational structures require an unambiguous description. A language to describe organisational structures was presented in Chapter 3, and basically consists of articulating the control regime of the organisation. The tasks of selecting an appropriate structure and describing it can be alleviated by re-using *organisational patterns*, discussed in Section 5.4.2.1.

Regarding their content, the design models take into account the selected organisational structure, which means, on the one hand, the appearance of new roles, protocols and organisational rules, and on the other hand, modifications of the elements already present in the preliminary models. Additionally, the design models must be complete and expressed in more detail than the analysis models. First, the role model must contain all the roles in the system, and each role must be completely defined. In particular, the *permissions* and *responsibilities* must be fully defined, since they specify the functionality of the roles. Second, the interaction model must include a protocol description for each of the interactions of the system. Lastly, the organisational model must include all the rules that govern the behaviour of the system, including those related to the organisational structure. In the case of this model, an important refinement is that the organisational rules must be expressed in the formal language defined in Section 3.3.4, rather than in natural language.

5.4.2 Activities

The activities in the organisational design are oriented towards establishing the structure of the system and towards detailing the interactions between the roles. The activities of the organisational design, and the order in which they are performed, are depicted in Figure 5.5. Also, the figure highlights the use of organisational patterns in determining the structure of a system. These activities are described in the following.

5.4.2.1 Defining the organisational structure

One of the main activities during the organisational design is precisely that of selecting an appropriate organisational structure for the system. Once selected, the organisational structure determines the information needed to complete the role and interaction models initiated during the analysis. This organisational structure is selected on the basis of several factors. First, although not all the communication paths are defined in the analysis models, the available paths provide a good insight into the topology. Second, the real-world problem modelled by the system typically possesses an integral organisational structure, which can be inherited in the system itself. Third, although the consideration of non-functional requirements for the system is out of the scope of this process, it is worth mentioning that different organisational structures exhibit different degrees of efficiency and robustness. Finally, other factors to consider when selecting

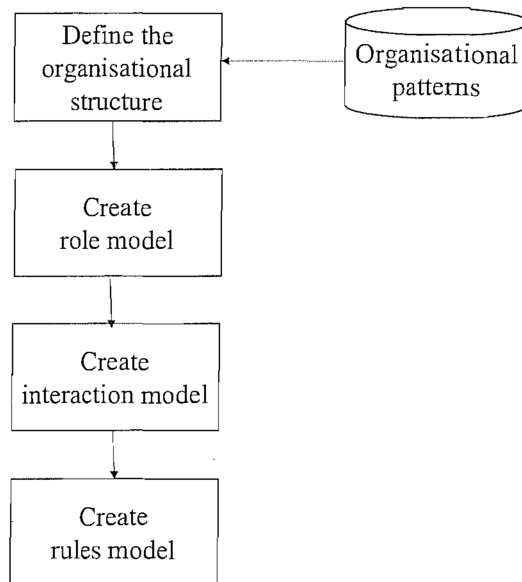

FIGURE 5.5: Activities of the organisational design

an organisational structure deal with facilitating the implementation and monitoring of organisational rules. Depending on the particular rules, some structures can facilitate this more than others.

Although it is difficult to give a detailed procedure to select the best organisational structure, we can provide some guidelines. First, the topology is depicted. The way this is done was illustrated in Figure 3.7, by representing roles as nodes and, for each protocol, an arc from the initiator to every collaborator. Next, the control regime is obtained and depicted. For this purpose, each protocol is analysed to determine the control relationship between the participants (initiator and collaborator). Such an analysis can be based on the nature of the interaction, on what is stated in the requirements, or by consulting an expert in the domain. If there is an authority relationship from one role to the other, this is depicted by means of an arrow in the arc.

After this, there are three possible situations. The first and simplest is when the designer *identifies* the organisational structure from the graphic. This assumes that the designer has previous experience and that the graphic is sufficiently descriptive. The second case consists of comparing the graphic to the structures included in a *catalogue of organisational patterns*, and selecting

the one that best matches. Such a catalogue was presented in Section 3.6, together with a procedure to select an appropriate pattern. The last case consists in building a new organisational structure, for which the graphic can be completed with additional roles and interactions, and then expressed in the language described in Chapter 3.

5.4.2.2 Creating the organisational design models

As mentioned above, the design models — the role model, the interaction model and the organisational rule model — are built from the corresponding analysis models. Specifically, for the interaction model each protocol description is checked to be consistent with the recently selected organisational structure. For example, it might be that a collaborator in a given protocol no longer exists, as a consequence of being joined with other roles or disappearing. In case of inconsistency, the necessary modifications are carried out. After this, any additional interaction, caused by the organisational structure, must be represented with a protocol description.

For the role model, each of the role descriptions in the preliminary role model is reviewed to be sure that it complies with the organisational structure. Some possible causes of non compliance are that the role is no longer part of the system, that the role no longer participates in a protocol, or that the role needs to be involved in other protocols. After this, it is checked that all the roles in the organisational structure have their corresponding role description in the model and, if not, they are created. Note that the description of the roles at this point must be complete, including exhaustive lists of permissions and responsibilities, the former expressed in the language proposed in the Gaia methodology [134].

Lastly, the organisational rule model is created from the preliminary organisational rule model. Again, the existing rules must be checked and adapted, if necessary, to the introduced organisational structure. Some reasons for adapting the rules are that some elements — roles or protocols — no longer exist, that they do exist but their meaning has changed, or that the rules have to involve new elements. Then, the rules governing the operation of the structure — those independent of the domain — are incorporated into the model, taking care that they are not in conflict with existing rules. Next, those rules that could have arisen as a result of the introduction of the organisational structure are added to the model. The last step in this activity is to express all the rules in the model in the language described in Section 3.3.4.

5.5 Agent design

During the organisational design, agents are considered to be black boxes, and their detailed composition is ignored. In contrast, during the agent design workflow — to which this section is devoted — the emphasis is placed on the internal design of the agents. In this way, the objective of agent design is to produce a specification of how each agent fulfils its requirements.

When detailing the internal structure of an agent, it must be noted that a huge variety of agents exists. For instance, there are agents that rely on close interaction with humans, while others act mainly with no human intervention. Also, some agents exhibit complex reasoning, whereas others exhibit such a simple behaviour as providing services on request. These varied types of agents differ in the details of their design; for example, agents with intense human interaction might need special consideration in designing human interfaces. Since it would be difficult to encompass all the possibilities in just a single method, in the following we provide guidelines for the most common aspects of agent design.

The agent design workflow takes its input from the organisational design, and its outputs are used in the next workflow, namely implementation. We describe below the artefacts and activities involved in the agent design, using the object paradigm as an approach to design systems.

5.5.1 Artefacts

There are two artefacts involved in the agent design: the *structure model* and the *functionality model*. Since we use the object approach as the means to design agents, the artefacts are also based on common object artefacts. The structure model provides a structural decomposition of a role into components, while the functionality model specifies how these components interact to achieve the desired role behaviour. We detail both artefacts below.

5.5.1.1 The structure model

The objective of the structure model is to decompose the design of each role into manageable units, or classes. This decomposition is done in terms of data and functionality. More specifically, the structure model is formed of class diagrams [117], one for each role in the system. Although a class diagram is a common concept in object-based techniques, it has several meanings and purposes — depending on the stage of the process in which is used — so it is worth explaining the way in which we use it here. In the structure model, we use a class diagram to describe the main internal components of a role (as classes), and the static relationships between them, such as *dependence*, *part-of* and *inheritance*. The level of detail in the description must be sufficient to identify the core classes, and for each of these classes, the operations necessary to achieve the functionality of the role, and the internal information required to implement these methods (attributes). However, it is not necessary that the diagram includes all the classes needed to implement the role, nor all the information and operations to implement each class. Thus, this level of detail required is that usually found in the design phase of methodologies such as RUP [117].

An example of a class diagram, corresponding to a fragment of a BDI agent, is shown in Figure 5.6. This class diagram contains five classes (*Agent*, *Intention*, *Event*, *Selector* and *ExternalEvent*), each of which is divided into three parts, containing their name, attributes (information), and operations (functionality). Between the *Agent* and *Selector* classes there exist a

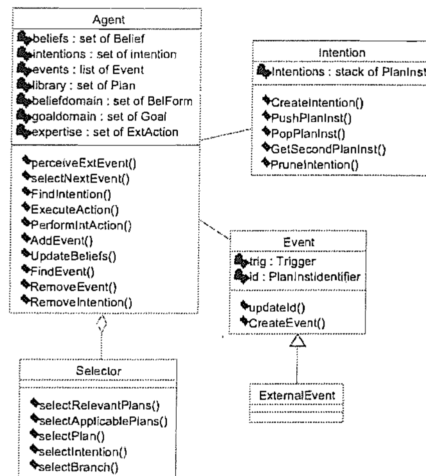

FIGURE 5.6: Example of a class diagram in the structure model

part-of relationship (indicated by the diamond and line), whereas between the *Event* and *ExternalEvent* classes there exist an *inheritance* relationship (indicated by the triangle and line). The fact that the *Agent* class uses the *Event* class for its operation, is indicated by a line linking these classes.

5.5.1.2 The functionality model

The functionality model consists of a set of *scenarios*, each of which belongs to a role, represents a piece of functionality of the role, and contains a sequence diagram showing how the role executes the functionality. Figure 5.7 depicts a generic functionality model, and the composition of the scenarios. The classes involved in the sequence diagram are those of the class diagram corresponding to the role. For example, in a market application, a possible scenario for the buyer role would represent the functionality *find the best price seller for a given product* by means of a sequence diagram showing how the classes of the buyer interact to achieve it.

The artefacts of the agent design workflow are illustrated in Figure 5.8, which shows that any role in the system, such as *Role i*, has an associated class diagram and a number of scenarios, each referring to a piece of functionality and described by a sequence diagram.

5.5.2 Activities

The activities involved in the agent design consist of selecting an appropriate agent architecture and of building the artefacts described above. These activities are illustrated in Figure 5.9,

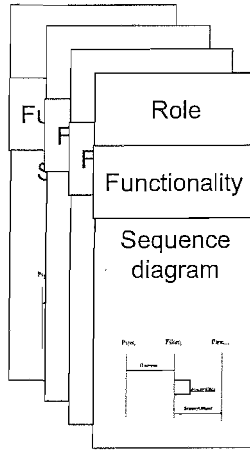

FIGURE 5.7: The structure model

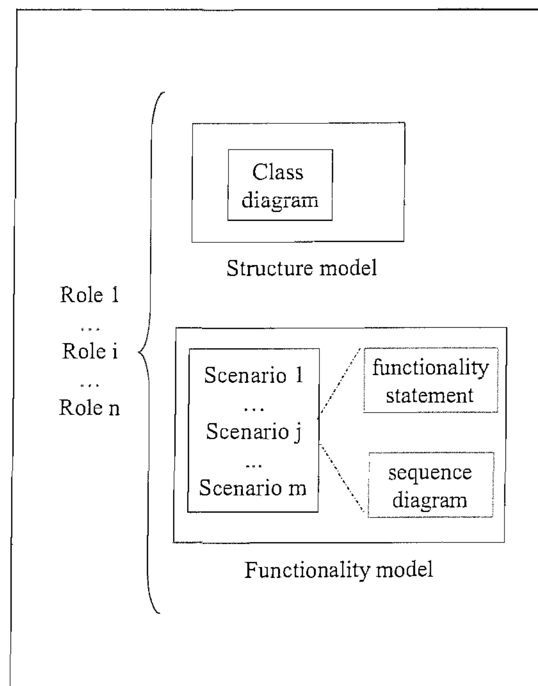

FIGURE 5.8: The artefacts of the agent design

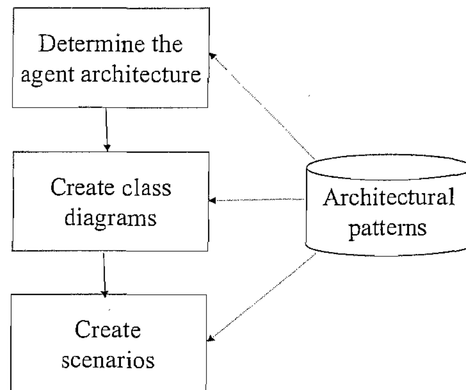

FIGURE 5.9: The activities of the agent design

together with indications of those stages in which the use of architectural patterns is useful. In this way, for every role in the system, the following activities are performed.

5.5.2.1 Determining the agent architecture

In this activity, the architecture for the role is determined. In order to do this, several factors must be considered. The most important factor deals with the degree of behaviour complexity expected from the role. For example, simple behaviour can be more easily implemented through reactive architectures, whereas complex behaviour may requires the use of deliberative or hybrid architectures. Another factor that determines the architecture deals with the level of pro-activeness required. Reactive architectures typically produce agents which are not pro-active, but operate only on request of other agents. On the other hand, BDI architectures are suitable for constructing highly pro-active agents. Other factors that affect the decision are the level of familiarity that the developers have with a specific architecture, and the support that different development tools provide to specific architectures. The accomplishment of this activity can be considerably facilitated by means of a catalogue (as the one presented in Chapter 4) showing, for each architecture, its characteristics, advantages, limitations and applicability.

5.5.2.2 Creating class diagrams

In order to create the class diagram, two different methods can be used. The first is to employ an object-based methodology such as RUP [117]. The second consists of using a catalogue of

architectural patterns, such as the one presented in Chapter 4, which also presents guidelines for selecting an appropriate pattern for a specific role.

Regardless of which method is used to construct the class diagram, the inputs are taken from the organisational design models. Specifically, the role model provides three inputs: the liveness responsibilities (which describe the functionality that the role is expected to exhibit), the safety responsibilities (describing the conditions that must hold during the lifetime of the role), and the permissions (which contain the environmental entities employed by the role, together with the rights to access them).

Additionally, the interaction model provides the inputs and outputs of the protocols in which the role participates, the organisational structure provides the control relationships involving the role, the organisational rule model provides the rules that constrain the behaviour of the agent and, finally, the services model provides the services of the role.

5.5.2.3 Creating scenarios

Similarly to the class models, the scenarios can be obtained by following an object-based methodology, or by using a catalogue of architectural patterns such as that in Chapter 4 which, apart from containing a procedure to select the appropriate pattern, also contains the main scenarios of the role functionality. When following an object-based methodology, it is advisable to decompose the functionality of the role by means of *use cases*, and then build the corresponding sequence diagram for each of them.

The inputs for creating the set of scenarios are the same as for creating the class diagrams, with the addition of the class diagrams themselves.

5.6 Implementation

Although implementation is not covered in detail in this thesis, it is important to provide some general guidelines mainly in terms of how it interfaces with the design and detailed design workflows.

Basically, the implementation consists in refining the models obtained in the organisational design and agent design to the extent that they can be directly implemented. However, in order to carry out this refinement, an important decision must be taken at this stage, particularly in relation to the implementation platform to be used. Several types and instances of agent-based platforms exist to date, some of the most popular being based on object technology, such as JADE and FIPA-OS.

An agent can play one or more roles, and it is during implementation that the decision is made about which agents will implement which roles. In principle, however, the fact that a role

is implemented in an agent does not ensure that the agent will actually play the role in the system, or during all its life cycle, since temporal assignments of roles to agents is possible in some platforms. The decision of which roles will be implemented by which agents must consider aspects such as physical distribution, sub-modules shared by different roles, efficiency, and cohesion of functionalities.

In the case that more than one role is implemented in a single agent, common sub-modules must be identified so that they do not have to be programmed more than once. For example, if two roles designed according to the same architectural pattern are implemented in the same agent, the classes that implement the functionality of the architecture need to be programmed just once and shared for the two roles.

Regarding the use of architectural patterns, it is important to note that in this thesis they are used as *design* patterns, as a tool during the design activities. In addition, similar patterns can be used to support the implementation. These patterns would differ in the level of detail that the classes and sequence diagrams contain, and can even include some form of automatically generated code. The construction of such patterns, however, is out of the scope of this thesis. Finally, although not referred to as implementation patterns, the examples included in implementation platforms can serve as a base from which specific applications can be constructed.

5.7 Iterations

The development cycle is divided into iterations. Each iteration encompasses all the workflows, from requirements analysis to implementation (other workflows such as transition and testing are not considered in this thesis). In this way, each iteration delivers an executable version of the system, and adds information to the artefacts of each workflow. We can group iterations into two parts, or phases: elaboration and construction. During the elaboration phase the emphasis is put on understanding the system and creating a stable architecture, while during the construction phase the efforts are directed towards accomplishing the functionality of the system.

While the decomposition into workflows is common for all applications, iteration decomposition varies from application to application, in terms of work dedicated to each workflow, number and, more importantly, purpose. As stated previously, early iterations generally dedicate more work to requirements and analysis, while later iterations dedicate more work to design and implementation. Also, as a general rule, the larger the system, the more iterations are needed. The actual decomposition of the development cycle into iterations is guided by the functionality of the system. This means that the functionality of the system is divided into parts, one or more of which are assigned to an iteration, whose purpose is to accomplish that part of the functionality. The order in which the iterations must be carried out is important and must be established as part of the iteration decomposition, since the most critical and important parts of functionality must be considered first, to obtain earlier user feedback and foresee possible changes in delivery times.

The following are helpful guidelines for iteration decomposition.

- The set of iterations must cover all the functionality expected from the system.
- The early iterations in the decomposition must be occupied by those functionalities that form the core of the system, or by those functionalities that involve a high risk of creating the wrong system or delaying the delivery of the system. An example of the former risks are processes that are critical but poorly described, while an example of the latter risk is the employment of new technology.
- Early iterations must provide the insight of most of the system.
- It is desirable to achieve a balance in the iterations, so that no iteration is too big nor too small.

Thus, the criteria to decompose the development into iterations is based on the division of the system by its intended functionality. This division can be facilitated by means of *use cases*, and dedicating one iteration to fulfil one or more use cases. Among all the use cases, the first selected are those that form the core functionality of the system or that may involve critical aspects that may lead to unforeseen situations, for example the development of a complex algorithm or the use of a new technology.

5.8 Related work and conclusions

Only few attempts have been made to employ the incremental iterative approach in agent-base methodologies. Among these attempts, MESSAGE [78] was one of the first agent-based methodologies to include an incremental iterative approach in its process. In general, the idea behind MESSAGE was to extend existing object-based methodologies to the agent paradigm. As a consequence, the process of MESSAGE closely resembles the process of RUP [69] (which is one of the most popular methodologies based on the object paradigm, and is based on the Unified Modelling Language (UML) [43]).

More recently, the incremental iterative approach process has been included in the process of the INGENIAS methodology [54], which itself takes several of its characteristics from MESSAGE, so it followed the same idea of adapting the RUP process. To put this idea into practice, some associations were established between object and agent concepts; for example, *class* was associated with *agent*, and *architecture* with *organisation*. These associations are natural, since — in the object paradigm — classes represent entities that encompasses the functionality of the system, and the architecture is what provides the system with a structure. From these associations, and using some previously defined *meta-models*, a list of activities was derived and grouped into workflows and phases. These meta-models are arguably what characterises INGENIAS,

and consist of diagrams of elements and their relationships, which are the components that a practitioner can employ to model a system.

Since the process presented in this chapter is also inspired by the RUP process, it is similar to the processes of INGENIAS (and, as a consequence, of MESSAGE), mainly in terms of structure. However, there are also several differences: while the INGENIAS process is closely based on object concepts, our process is based on the Gaia methodology, which is a genuine agent-based methodology in itself. One consequence of this is that we use the graphical models provided by Gaia (in the analysis and organisational design workflows), while INGENIAS relies on purpose-built meta-models. Another difference is that of coverage, since our process is focused on analysis and design, and INGENIAS' considers, in addition, implementation and testing. Finally, the INGENIAS process does not explicitly facilitate the development of open systems, since it uses purpose-built models and gives no insight into how other well known models or architectures can be adapted to fit into the process. In contrast, our process relies on common abstractions, on easily-adaptable models, and considers the use of different agent architectures.

It is worth mentioning that, although not explicitly stated, the process presented in this chapter can also be used in the construction of new agents that are incorporated into an existing open system, taking into account the considerations described below. First, the analysis and design of the overall system have already been undertaken, and their results must be available to the developers of the new agents. At this point there are two possibilities. One possibility is that the existing system has been designed according to the process presented in this chapter, or a similar one (for example Gaia). In this case, the results of the design can take the form of the design models presented in this chapter. The second possibility refers to the case in which the existing system was designed using a completely different process. In this case, a specification of the system which is independent of the process is needed (a specification such as this, based on organisational concepts, is presented in the next chapter). Second, a decision has to be made about the roles that a new agent will implement. Once the roles have been determined, the developer can re-use the models of the agent design workflow corresponding to those roles (if available), or can produce new designs.

Chapter 6

Case study

The previous chapters contain isolated examples that illustrate and clarify the contributions of this thesis but, being isolated, they focus only on their stand-alone usage. However, most of the benefit of our contributions relies on their use in the context of an overall methodological process. To illustrate this, in this chapter we describe the development of a system using the methodological process presented in Chapter 5, which encompasses most of the contributions of this thesis. Most importantly, this case study provides evidence of the benefits of using this development process, and its different parts.

The system selected for this case study is easy to understand, has a reasonable number of roles and interactions, and is *open*; that is, it allows the incorporation of components not known at design time. In addition, the problem statement of this system was extracted from the literature of agent-oriented methodologies, which has the benefits of facilitating comparison of our process with others and, most importantly, it offers a demonstration of its value for a standard problem, rather than one tailored to our solution.

The rest of this chapter is organised as follows. In Section 6.1, we describe the problem statement of the system. Then, in Section 6.2, we present a decomposition of the development into iterations. The activities performed, and the artefacts produced, during the first and second of these iterations are presented in Section 6.3 and Section 6.4, respectively. Finally, our conclusions are presented in Section 6.5. Note that, although the development of this system requires several iterations, only the first two are described, since the others are very similar, and their inclusion would not add any benefit to the purpose of this chapter.

6.1 Problem statement

The case study considered in this chapter is based on an example presented in [96], where it is used to illustrate the INGENIAS methodology, and deals with the segmentation of users according to common interests. The following describes the problem statement of the system.

A new system is required for segmenting users into groups of common interest. The system-to-be will complement the services provided by a commercial Web, and will be used for marketing purposes, such as offering specific products only to users in a group with related common interest. The system is conceived as a multi-agent system in which each (human) user is represented by a *personal clerk*, which groups with other personal clerks to form a community. Such a community is represented by a *clerk of community*, and relates to one subject. Such a segmentation of interests helps to control the quality of documents provided to users, as explained below.

A community can be seen as a source of information to which users subscribe to obtain relevant information for their interests. Once subscribed, a user begins to receive information from the community. This information originates from members of the community or from other sources of information not specified. The information that the users receive passes through a series of filters to ensure its quality.

When a user suggests information to the community, the community first compares the suggestion with the *community profile*. If the information matches the community profile, the document is evaluated by a set of members of the community. However, before being evaluated by their users, each of their personal clerks decides, on their own, whether the document is interesting to its user. In the affirmative case, the evaluation request is presented to the user, so that he evaluates the document. In the negative case, a vote against the document is produced. The suggested document is approved only if most of the consulted members voted in favour of the document, and the positive and negative evaluations are registered and used in the acceptance of future suggestions.

The permanence of members in a community is subject to the following restrictions.

- Users who have suggested many documents evaluated negatively are expelled, since their interests are not in accordance with those of the community.
- Users who evaluate too many documents negatively are also expelled, since they have not shown interest in the type of information provided by the community.

Community clerks and personal clerks describe their interests by means of a *profile*, which can take the form of a set of documents (the last documents evaluated positively), keywords or categories. The keywords and categories of a clerk can be modified by its user.

The system to be developed must admit the incorporation of new sources of information, such as news forums of news and other communities. For example, the news published in a forum can be valuable for the members of a community whose interests are similar. In addition, different communities can collaborate to exchange information. The exchange of information must be allowed as long as it has been properly authorised by the administrator of the system. Both mechanisms

are used to supply information to communities, overcoming a possible passivity of users.

Users connect with their clerks by means of a Web interface that allows them to: suggest documents, evaluate documents, see documents, and see statistics of operation.

Additionally, an administrator of the system is in charge of:

- creating new communities of users;
- eliminating communities with low numbers of members;
- eliminating users who have been inactive for too long; and
- configuring parameters of execution of the agents, such as the maximum number of agents that can exist in the system, thresholds of document acceptance, times that a user can negatively evaluate a document before being ejected from the community, and the number of users that evaluate a document.

It should be clear that this system is inherently open, since it must allow the incorporation of users (or personal clerks) not known at run-time, as well as of information sources. In the following, we show the application of the methodological process described in Chapter 5 to this problem.

6.2 Iterations

The functionality of the system can be divided into the following parts.

Part 1 Approve new information: the process of receiving, filtering, and disseminating documents suggested by users.

Part 2 Exchange information: the part dealing with the exchange of information between different communities or other sources of information.

Part 3 Create communities: the process of creating new communities in the system.

Part 4 Eliminate communities: the elimination of unwanted communities from the system.

Part 5 Register new users: the process of accepting new users in the communities.

Part 6 Expel users: the part dealing with expelling unwanted users from communities.

This partition is used as basis for the iteration decomposition of the system, which also takes into account the following two factors. First, it considers the potential size of each of the parts, and tries to keep a balance in the sizes of the iterations. Second, it prioritises the parts by their importance in the functionality of the system. In particular, it recognises *Part 1* as the core of

Iteration	Parts	Functionalities
1	Part 1	Approve new information
2	Part 5 and Part 6	Register new users and expel users
3	Part 3 and Part 4	Create communities and Eliminate communities
4	Part 2	Exchange information

TABLE 6.1: Iteration decomposition of the case study

the system, since it directly supports the accomplishment of the goal of the system, and is also the most complex part, involving several components of the system. The decomposition of the system into iterations is presented in Table 6.1, and the next section describes the first of these iterations.

6.3 First iteration

As was established previously, the first iteration deals with approving new information to the system, which means receiving, filtering, and disseminating documents suggested by users. Consequently, in this section we show the results of applying the analysis, organisational design and agent design workflows to this part of the functionality of the system.

6.3.1 Analysis

As was discussed in Section 5.3, the analysis workflow consists of the elaboration of five artefacts: sub-organisation model, environment model, preliminary role model, preliminary interaction model, and preliminary rules model, each of which is described below.

6.3.1.1 Organisation model

Because of its small size, this system need not be decomposed into sub-organisations. However, for purposes of exposition, we can say that a possible decomposition could be structured into four sub-organisations: the components around the human user, including personal clerks; the sources of information; the filters of information; and the components in charge of administering the system.

6.3.1.2 Environmental model

We divide the entities in the environmental model into two types: resources and external agents, the former of which are entities composed only of information. The information in the resources can be accessed or modified by the agents in the system that have the corresponding rights. In

our environmental model, the resources correspond to the tuples of the Gaia environmental model [134].

However, in our model we also contemplate entities more complex than resources. The external agents are entities for which a description based just on tuples of information would be unnatural or very coarse. This type of entity encapsulates not only data but also behaviour, expertise, autonomy or pro-activeness, and includes humans, non-trivial legacy software, and multi-agent systems as the most representative examples. Similarly to agents in the system, external agents' functionality can be decomposed into roles. It must be noted that, although external agents and agents are similar, external agents do not belong to the system and it is only their *interaction* with the system that is important and not their *internal composition*. The resources for this first iteration are the following.

- Document: the key piece of information in the system, formed of a title, authors, keywords and body.
- Profile: expresses document interests of a clerk, either personal or of a community.
- Evaluation: contains information about the acceptance of a document.
- Vote: expresses approval or rejection of a document by a user (or his clerk).

The external agents for this first iteration are the following.

- Reader: a human user interested in accessing a document, presumably for reading.
- Voter: a human user that shows his opinion about accepting a document.
- Recommender: a human user that proposes a document.

6.3.1.3 Preliminary role and interaction models

The roles necessary to accomplish the functionality of this iteration are obtained directly from the problem statement and consist of: *PersonalClerk*, *CommunityClerk*, *Profiler*, *Evaluator* and *Broadcaster*.

The interaction protocols are shown in Figure 6.1, in which roles have been represented by ovals, external agents by rectangles and protocols by solid arrows. Additionally, dashed arrows indicate protocols whose participants have not yet been determined.

Based on this information, a preliminary description of the roles is obtained, as shown in Figure 6.2, Figure 6.3, Figure 6.4, Figure 6.5 and Figure 6.6. Note that each role representation contains the name of the role, a brief description, its permissions, and its responsibilities. However, at the preliminary role model, it is not necessary that all the roles are completely described.

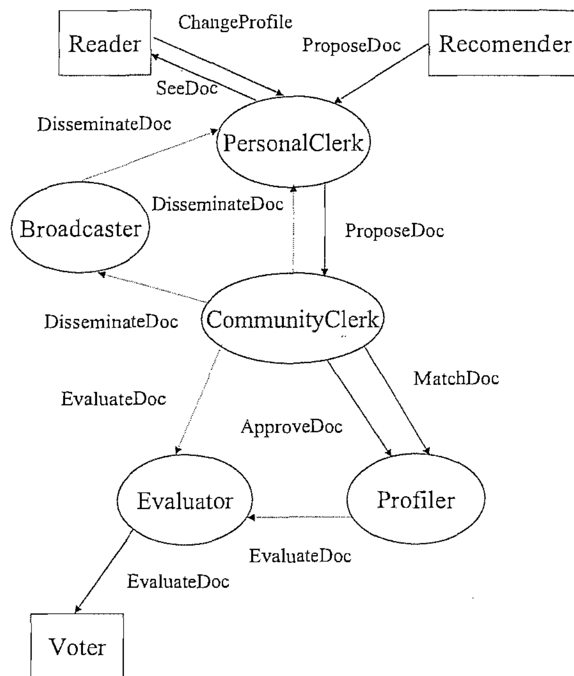

FIGURE 6.1: Communication paths in the case study

Role Schema:	PersonalClerk
Description:	Represents a human user
Protocols and Activities:	SeeDoc, ProposeDoc, DisseminateDoc
Permissions:	
Responsibilities:	
Liveness:	
Safety:	

FIGURE 6.2: Preliminary roles, part 1

Role Schema:	CommunityClerk
Description:	Represents a community of users
Protocols and Activities:	ProposeDoc, EvaluateDoc, MatchDoc, ApproveDoc, DisseminatDoc
Permissions:	
Responsibilities:	
Liveness:	
Safety:	

FIGURE 6.3: Preliminary roles, part 2

Role Schema:	Evaluator
Description:	Decides if a document is interesting to a particular user
Protocols and Activities:	EvaluateDoc, <u>DecideIfDocInter</u>
Permissions:	
Responsibilities:	
Liveness:	
Safety:	

FIGURE 6.4: Preliminary roles, part 3

Role Schema:	Profiler
Description:	Decides if a document is relevant to a community
Protocols and Activities:	MatchDoc, ApproveDoc, <u>matchProfileDoc</u>, <u>CountVotes</u>
Permissions:	•
Responsibilities:	
Liveness:	•
Safety:	•

FIGURE 6.5: Preliminary roles, part 4

Role Schema:	Broadcaster
Description:	Disseminates a document into a community
Protocols and Activities:	DisseminateDoc
Permissions:	•
Responsibilities:	
Liveness:	•
Safety:	•

FIGURE 6.6: Preliminary roles, part 5

SeeDoc	
PersonalClerk	Reader
The personal clerk presents a new document to its user	

FIGURE 6.7: Preliminary interaction protocols, part 1

ProposeDoc	
Recommender	PersonalClerk
A user suggests a document to the community	

FIGURE 6.8: Preliminary interaction protocols, part 2

DisseminateDoc	
?	PersonalClerk
An approved document is sent to the members of the community	

FIGURE 6.9: Preliminary interaction protocols, part 3

In addition, the preliminary interaction model contains the main interaction protocols, and their most significant characteristics. The protocols are taken from the role representations of the preliminary role model. Figures 6.7, 6.8, 6.9, 6.10, 6.11, 6.12, and 6.13 show the protocols for this iteration. Note that in each protocol description, the upper box contains the name, the lower box the description, the middle boxes the initiator and collaborators, and the inputs and outputs are located outside the boxes. A question mark in the boxes of the initiator or collaborators indicates that there is more than one possibility, and that the decision is postponed until the design workflow.

6.3.1.4 Preliminary rule model

The following are the rules that control the operation of the system. Note that liveness rules are rules that the agents try to bring about, while safety rules are those that the system must avoid violating. Note also that, in the *preliminary* rule model, organisational rules are described just in plain English.

EvaluateDoc	
?	Evaluator, Voter
A document is evaluated by a user and its clerk	

FIGURE 6.10: Preliminary interaction protocols, part 4

MatchDoc	
CommunityClerk.	Profiler
A document is checked against the profile of a community	

FIGURE 6.11: Preliminary interaction protocols, part 5

ApproveDoc	
CommunityClerk	Profiler
A decision is taken about accepting or rejecting a document	

FIGURE 6.12: Preliminary interaction protocols, part 6

ChangeProfile	
Reader	PersonalClerk
The user changes his profile	

FIGURE 6.13: Preliminary interaction protocols, part 7

The liveness organisational rules for this iteration are as follows:

- A suggested document is first compared to the community's profile; then it is evaluated by personal agents of the community and finally it is voted on by the users.
- A suggested document must be evaluated by at least five users (and is approved if it has been approved by a majority of its evaluators).

In addition, the safety rules for this iteration are as follows:

- A user cannot suggest the same document more than once.
- The same document must not be disseminated to a user more than once.

6.3.2 Organisational design

In the following we describe the application of the organisational design to the first iteration of the system.

6.3.2.1 Organisational structure

In order to determine an organisational structure that fits the system, we note that the communication paths of the system, depicted in Figure 6.1, do not resemble any common topology. However, after considering the nature of the protocols and arranging the roles, the structure shown in Figure 6.14 is obtained. If this structure is compared to the patterns of the catalogue presented in Section 3.6, we can note that it is similar to the simple hierarchy, except for two differences. The first difference is that, in the pattern, there are no communication paths between members of the lower level, while in the structure of this case study, there is a communication path between the *Broadcaster* and the *PersonalClerk*, both members of the lower level. The second difference is that, in the pattern, all the control relationships from the head to the leaves are *authority* relationships, while in this case study there is a *peer* relationship between the *PersonalClerk* and the *CommunityClerk*.

At this point, there are two alternatives: to consider the organisational structure either as a special type of hierarchy, or as a completely different structure. More generally, the alternatives are: to adapt the problem to the pattern in question, or not to use any existing pattern at all (in which case a new pattern can be constructed). This situation is not uncommon since it can be the case that no pattern in a catalogue suits completely a given application. The selection of an alternative largely depends on each particular situation, but should be based on what is stated in the pattern, specifically in the *forces*, *restrictions* and *consequences* sections.

In this case study, we assume that the alternative of adapting the problem is taken, and then we adjust the preliminary role and interaction models to make the structure more similar to a

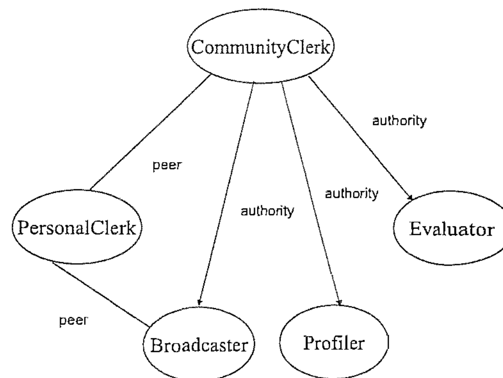

FIGURE 6.14: Preliminary organisational structure

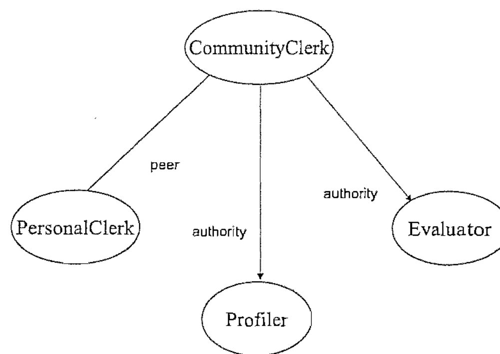

FIGURE 6.15: Organisational structure

simple hierarchy, thus yielding the structure depicted in Figure 6.15. The modification consists in merging the *CommunityClerk* and *Broadcaster* roles, so that the *peer* relationship between *PersonalClerk* and the *Broadcaster* is eliminated.

6.3.2.2 Role model

As a consequence of the introduction of this organisational structure, the list of roles in the system has changed to: *PersonalClerk*, *CommunityClerk*, *Profiler* and *Evaluator*. Note the absence of the *Broadcaster* preliminary role, and that external agents are not modelled as roles, since only their interactions are relevant. In the role model, the descriptions of the roles are completed, resulting in the role descriptions shown in Figure 6.16, Figure 6.17, Figure 6.18 and Figure 6.19.

Role Schema:	PersonalClerk
Description:	Represents a human user
Protocols and Activities:	SeeDoc, ProposeDoc, DisseminateDoc, UpdateProfile
Permissions:	**reads** document **changes** profile
Responsibilities:	
Liveness:	PersonalClerk=(ProposeDoc | DisseminateDoc | (ChangeProfile . UpdateProfile))^{w}
Safety:	•

FIGURE 6.16: Role model, part 1

Role Schema:	CommunityClerk
Description:	Represents a community of users
Protocols and Activities:	ProposeDoc, EvaluateDoc, MatchDoc, ApproveDoc, DisseminatDoc
Permissions:	**reads** document, evaluation, vote **changes** profile
Responsibilities:	
Liveness:	CommunityClerk = (ProposeDoc | EvaluateDoc | MatchDoc | ApproveDoc, DisseminatDoc)^{w}
Safety:	•

FIGURE 6.17: Role model, part 2

Role Schema:	Evaluator
Description:	Decides if a document is interesting to a particular user
Protocols and Activities:	EvaluateDoc, DecideIfDocInter
Permissions:	**reads** document, profile **changes** vote
Responsibilities: Liveness:	Evaluator = (EvaluateDoc . DecideIfDocInter)^{w}
Safety:	•

FIGURE 6.18: Role model, part 3

Role Schema:	Profiler
Description:	Decides if a document is relevant to a community
Protocols and Activities:	MatchDoc, ApproveDoc, matchProfileDoc, CountVotes
Permissions:	**reads** document, profile **changes** evaluation.
Responsibilities: Liveness:	Profiler = ((MatchDoc . matchProfileDoc) | (ApproveDoc . CountVotes))^{w}
Safety:	•

FIGURE 6.19: Role model, part 4

ProposeDoc		
Recommender	PersonalClerk, CommunityClerk	
A user suggest a document to the community		document
		nil

FIGURE 6.20: Interaction model, part 1

DisseminateDoc		
CommunityClerk	PersonalClerk, Reader	
An approved document is sent to the members of the community		document
		nil

FIGURE 6.21: Interaction model, part 2

EvaluateDoc		
CommunityClerk	Evaluator, Voter	
A document is evaluated by a user and its clerk		document
		vote

FIGURE 6.22: Interaction model, part 3

MatchDoc		
CommunityClerk.	Profiler	
A document is checked against the profile of a community		document, profile
		vote

FIGURE 6.23: Interaction model, part 4

6.3.2.3 Interaction model

The preliminary interaction model is modified to reflect the introduction of the organisational structure, resulting in the protocol descriptions shown in Figure 6.20, Figure 6.21, Figure 6.22, Figure 6.23, Figure 6.24 and Figure 6.25.

ApproveDoc		
CommunityClerk	Profiler	
A decision is taken about accepting or rejecting a document		Document, evaluation
		evaluation

FIGURE 6.24: Interaction model, part 5

ChangeProfile		
Reader	PersonalClerk	
The user changes his profile		nil
		profile

FIGURE 6.25: Interaction model, part 6

6.3.2.4 Rule model

The rule model expresses, in the LEVOR language, the organisational rules included in the preliminary rule model. The following are the liveness organisational rules, expressed in the LEVOR language (see Section 3.3.4).

$$card(PersonalClerk) \geq 1$$
$$card(Evaluator) \geq 1$$

There must be at least one evaluator and one personal clerk.

$$card(CommunityClerk) = 1$$
$$card(Profiler) = 1$$

There must be one and only one community clerk and profiler.

$$terminated(ProposeDoc)\ \mathcal{B}\ initiated(MatchDoc) \wedge$$
$$terminated(MatchDoc)\ \mathcal{B}\ initiated(EvaluateDoc) \wedge$$
$$terminated(EvaluateDoc)\ \mathcal{B}\ initiated(ApproveDoc) \wedge$$
$$terminated(ApproveDoc)\ \mathcal{B}\ initiated(DisseminateDoc)$$

Every suggested document must be filtered (compared to the community's profile, evaluated and voted) before it is disseminated.

$$\forall\ d : Document(\ card(EvaluateDoc(CommunityClerk, Evaluator, d)) = 5)$$

A suggested document must be evaluated by exactly five users.

Note the relationship between this rule and the one stating that there must be at least *one* Evaluator. The former refers only to evaluation of documents, while the latter applies to the whole operation of the system (including the case when there are no documents to be evaluated). However, an alternative design might state that there must be at least *five* Evaluators.

In addition, the safety rules for this iteration are as follows.

$$\forall p : PersonalClerk(\ \forall d : Document(\ card(ProposeDoc(p, CommunityClerk, d)) \leq 1))$$

A user cannot suggest the same document more than once.

$$\forall p : PersonalClerk(\ \forall d : Document(\ card(DisseminateDoc(CommunityClerk, p, d)) \leq 1))$$

The same document must not be disseminated to a user more than once.

6.3.3 Agent design

The agent design workflow consists of the design of each role of the iteration. Since the process is essentially the same for all the roles, we will describe here only the agent design for the *Profiler* role, since its simple functionality makes it suitable for exposition.

As can be observed in the organisational structure, the *Profiler* role is completely subordinated to the authority of the *CommunityClerk*. Additionally, according to the role model, the behaviour of the *Profiler* role can be described as a process of receiving orders, performing the activities related to accomplishing these orders, and replying with the results produced by the activities. Considering its reactive behaviour, and based on the catalogue of architectural patterns of Chapter 4, we conclude that the *Profiler* role can be modelled by means of the subsumption architecture [10]. Thus, using the subsumption architectural pattern, specifically the SOLUTION section (Section 4.3.2), and the corresponding procedure to obtain a detailed design (Section 4.3.3), we construct the fragments of the structure and functionality models corresponding to this role, as described below.

6.3.3.1 Structure model

The *Profiler* role interacts with the environment through two protocols: *MatchDoc* and *ApproveDoc*. According to this, the environment perceived by the *Profiler* can be described as the set of tuples, *(command, content1, content2)*, where: *command* is an identifier of the type of protocol (for example, *Match* for the *MatchDoc* protocol, or *Approve* for the *ApproveDoc* protocol); *content1* is a document; and *content2* is an *evaluation* if *command* is *Match*, or *nil* otherwise (this corresponds to the outputs of these protocols, as stated in the interaction model).

Accordingly, there are two behaviours for this role, as described below.

b1 *if (Match, d, e) is perceived then execute MatchProfileDoc(d) and continue the execution of protocol MatchDoc.*

b2 *if (Approve, d, e) is perceived then execute IsApproved(d, e) and continue the execution of protocol ApproveDoc.*

Here, *MatchProfileDoc* and *IsApproved* are activities of the *Profiler* role, dealing with matching a document to the community profile, and approving a document, respectively, as is stated in its role description.

To determine the inhibition relationship, it must be noted that, in this particular case, the inhibition relationship is irrelevant, since no perceived state can match both *b1* and *b2*.

The class diagram for the structural model is obtained by enhancing the class diagram of the subsumption pattern, with the particular characteristics of the *Profiler* role, resulting in the diagram shown in Figure 6.26. As can be observed, the enhancements consist in the elimination of the original *Inhibitor* class (since no inhibition relationship is required), the description of the information perceived (*Percept* class), and the representation of the *Inhibitor* activities as actions of behaviours.

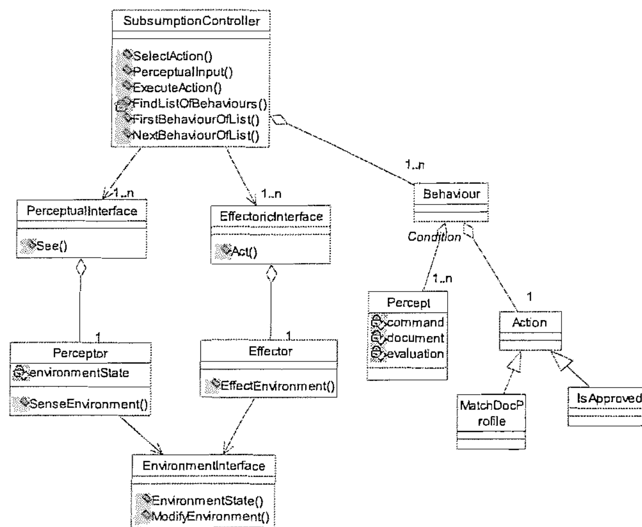

FIGURE 6.26: Class diagram of the structure model

6.3.3.2 Functionality model

The operation of the *Profiler* is so simple that only one scenario is needed to describe its functionality. Such scenario describes the dynamics followed by the classes to accomplish the functionality of the role, and is expressed by a sequence diagram adapted from the subsumption pattern. This sequence diagram, which is shown in Figure 6.27, is easier to interpret if we consider that the *Profiler* perceives the environment by receiving messages and interpreting their content, and affects the environment by sending messages.

In a real situation, the first iteration continues towards the development of an executable version of the system that implements the functionality of the iteration. This executable is evaluated by the stakeholders, and failures in satisfying the requirements are identified. Then, during the second iteration, in addition to implementing the corresponding functionality, changes are made to the appropriate deliverables (analysis and design models, and executables) in order to correct those failures. During the evaluation of the executable, other changes to the deliverables may arise due to modifications in the requirements. For example, in our case study, the stakeholders may realise that the organisational rule

> *A suggested document must be evaluated by exactly five users.*

can never be observed in communities with less than *five* users, which can cause them to modify the original requirements, and in turn the organisational rules model and the executable.

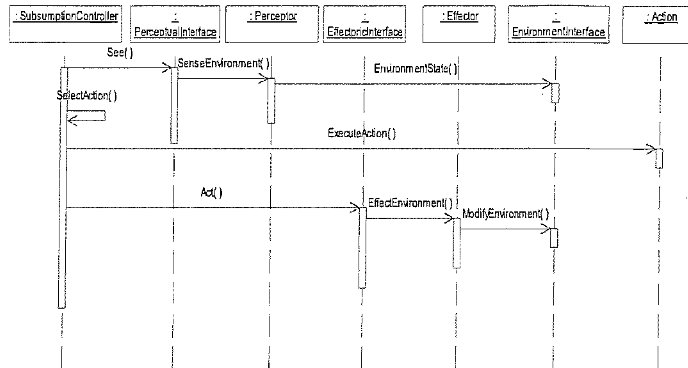

FIGURE 6.27: Sequence diagram of the functionality model

6.4 Second iteration

Once the first iteration has been concluded, the second iteration is started. The second iteration addresses another part of the functionality of the system, and consists of enhancing the results of the previous iteration in terms of adding elements to the artefacts, extending those elements or modifying them. For example, in the case of the preliminary role model, new roles can be added and existing roles can be modified to include the interaction with the new roles. In this case study, the second iteration deals with the registration of new users and the expulsion of inadequate users, as described below.

6.4.1 Analysis

In order to address the introduction of new users into the community and expulsion of users from the community, neither sub-organisations nor environmental entities need to be created. Similarly, no new roles are required, since this functionality can be carried out by the *PersonalClerk* and *CommunityClerk*. However, the incorporation of new protocols to the preliminary interaction model is required to cope with these tasks. Such protocol descriptions are shown in Figure 6.28 and Figure 6.29, the first of which refers to the registration of new users and the second to the expulsion of users.

In addition, the preliminary role model must be modified to incorporate these new protocols. Specifically, the descriptions of the roles involved in the protocols (the *CommunityClerk* and the *PersonalClerk*) must be updated, resulting in the descriptions shown in Figure 6.30 and Figure 6.31.

RegisterUser	
Reader	PersonalClerk, CommunityClerk
The user registers with the community	

FIGURE 6.28: Preliminary protocol description of registering users

ExpelUser	
CommunityClerk	PersonalClerk, Reader
Expels a user from the community	

FIGURE 6.29: Preliminary protocol description of expelling users

Role Schema:	CommunityClerk
Description:	Represents a community of users
Protocols and Activities:	ProposeDoc, EvaluateDoc, MatchDoc, ApproveDoc, DisseminatDoc, RegisterUser, ExpelUser
Permissions:	**reads** document, evaluation, vote **changes** profile
Responsibilities:	
Liveness:	CommunityClerk = (ProposeDoc | EvaluateDoc | MatchDoc | ApproveDoc, DisseminatDoc)^{w}
Safety:	

FIGURE 6.30: Preliminary role description of the community clerk

Role Schema:	PersonalClerk
Description:	Represents a human user
Protocols and Activities:	SeeDoc, ProposeDoc, DisseminateDoc, RegisterUser, ExpelUser
Permissions:	
Responsibilities: Liveness:	
Safety:	

FIGURE 6.31: Preliminary role description of the personal clerk

As for the organisational rules, the following rules regarding the expulsion of users must be incorporated.

- A user is expelled if he has suggested more than five documents evaluated negatively.
- A user is expelled if he has evaluated negatively more than five documents.

Note that we have arbitrarily chosen *five* as the number of documents to decide the expulsions.

6.4.2 Organisational design

During the organisational design of the second iteration, the organisational structure, as well as the role, interaction and rule models, are updated to reflect the changes introduced by the new protocols. In the case of the organisational structure, it can be observed that no changes are required, since the new protocols, *RegisterUser* and *ExpelUser*, do not alter the *peer* relationship between the *PersonalClerk* and the *CommunityClerk*.

However, some updates are necessary for the role, interaction and rule models, since, during the analysis, the corresponding preliminary roles were modified. In the case of the role model, the updates consist in modifying the *Protocols and Activities* section, as well as the liveness responsibilities of the descriptions of the *PersonalClerk* and the *CommunityClerk* roles, resulting in the descriptions shown in Figure 6.32 and Figure 6.33.

Role Schema:	PersonalClerk
Description:	Represents a human user
Protocols and Activities:	SeeDoc, ProposeDoc, DisseminateDoc, UpdateProfile , RegisterUser, ExpelUser
Permissions:	**reads** document **changes** profile
Responsibilities:	
Liveness:	PersonalClerk=(ProposeDoc | DisseminateDoc | (ChangeProfile . UpdateProfile) | RegisterUser | ExpelUser)^{w}
Safety:	.

FIGURE 6.32: Role description of the personal clerk

Role Schema:	CommunityClerk
Description:	Represents a community of users
Protocols and Activities:	ProposeDoc, EvaluateDoc, MatchDoc, ApproveDoc, DisseminatDoc, RegisterUser, ExpelUser, DecideExpel, Registry
Permissions:	**reads** document, evaluation, vote **changes** profile
Responsibilities:	
Liveness:	CommunityClerk = (ProposeDoc | EvaluateDoc | MatchDoc | ApproveDoc, DisseminatDoc | (RegisterUser . Registry) | (DecideExpel . ExpelUser))^{w}
Safety:	.

FIGURE 6.33: Role description of the community clerk

RegisterUser		
Reader	PersonalClerk, CommunityClerk	
The user registers with the community		profile
		acceptance

FIGURE 6.34: Protocol description of registering users

ExpelUser		
CommunityClerk	PersonalClerk, Reader	
Expels a user from the community		nil
		nil

FIGURE 6.35: Protocol description of expelling users

For the interaction model, the updates consist of completing the protocol descriptions of the *RegisterUser* and *ExpelUser* protocols, as shown in Figures 6.34 and Figures 6.35.

Finally, in order to update the rule model, the introduced organisational rules must be expressed in the LEVOR language, as follows.

$$(PersonalClerk.NOfNegSug() > 5) \Rightarrow initiates(CommunityClerk, ExpelUser)$$

A user is expelled if he has suggested more than five documents evaluated negatively.

$$(PersonalClerk.NOfNegEval() > 5) \Rightarrow initiates(CommunityClerk, ExpelUser)$$

A user is expelled if he has evaluated negatively more than five documents.

Note that in the expression of these organisational rules, we have assumed the existence of the activities *NOfNegSug* and *NOfNegEval* in the *PersonalClerk* role, which are used to obtain the number of suggested documents that are negatively evaluated by others, and the number of documents suggested by others that it evaluates negatively, respectively.

6.4.3 Agent design

The second iteration concludes by performing the agent design of each role introduced in the iteration, and by modifying the models for those roles who were affected by the introduction of new protocols and activities. However, in our particular case, no new roles were introduced during the second iteration. Similarly, the agent design for the *Profiler* role needs no modification, since this role was not affected by the protocols and activities introduced in the iteration.

The completion of this case study would require us to carry out the last two iterations, as was described in Section 6.2. This, however, would easily double the number of pages dedicated to this chapter and would add no benefit to its purpose, since the process would be essentially the same as that presented for the first two iterations.

6.5 Conclusions

In this chapter, we have presented the development of a case study using the methodological process described in this thesis, which includes the use of most of its contributions: the support of the organisational design by means of organisational patterns; the incorporation of agent design, based on agent architectures, into the development process; and the use of an iterative approach to make more agile the development process. In summary, the development consisted of decomposing the construction of the system into iterations, and accomplishing these iterations. Each iteration included the analysis, organisational design and agent design of a specific part of the system.

Although simple in concept, this case study illustrates the characteristics of the process and helps to draw some conclusions about its drawbacks and benefits. The most notable drawback is the lack of a software tool to support the construction of artefacts, which tends to make this task error-prone and burdensome. Also, the analysis of interactions, represented by the interaction model, lacks detail, since it does not consider the decomposition of the protocol into messages and, as a consequence, the decomposition of messages into communicative acts and content. Finally, specifically for the organisational pattern used (the simple hierarchy pattern), some lack of detail was found in the description of the roles involved.

In spite of these drawbacks, the case study showed that the process is straightforward, natural, and requires relatively little knowledge about agent-based computing for its use. Additionally, the incorporation of catalogues of patterns in key stages, reduces work, time, and alleviates the learning curve. More generally, the case study suggests that the enhancements we made to the basic process of the Gaia methodology (the incorporation of a catalogue of organisational patterns, the incorporation of the agent design phase, the use of architectural patterns to support this phase, and the decomposition of the process into iterations) are valuable and significantly increment its maturity.

Chapter 7

Specification and Integrity in the Development of Open Systems

7.1 Introduction

The number of computers and computational devices has increased significantly in the last few years and, since these devices rarely work on their own, the number of networks has also exploded. In software, new technologies such as the Internet, pervasive computing and the Grid are also emerging and take advantage of these networks. These technologies have brought challenging problems in computer science and software engineering, since they demand systems that are highly distributed, proactive, situated and *open*.

As was stated in Section 1.5, an open system is one that allows the incorporation of components at run-time whose internal composition may not be known at design time, but only their external functionality. The components of an open system may not be designed and developed by the same group, nor do they represent the same stakeholders. In addition, different groups may use different development tools and may follow different policies or objectives. Regardless of how and by whom a component is developed, it typically has the same rights to access the facilities provided by the system, as well as the obligation to adhere to its rules.

However, traditional approaches (e.g., object-oriented and component-based computing) have fallen short in engineering this type of application [91] because they operate at too low a level of abstraction. For example, object-oriented computing decomposes a system into entities (or *objects*) that encapsulate information and functionality. This information, however, usually refers to basic data structures or to other objects. Similarly, the functionality of objects relies on simple procedures like those normally found in most programming languages. Elaborated object decompositions, although possible, tend to make it difficult to understand and design applications that involve high-level concepts such as grid services and workflows.

In response, different approaches have been attempted to facilitate the development of such complex applications. In particular, some evidence suggests that the multi-agent approach provides adequate abstractions to successfully develop this type of system [75], and this has resulted in the appearance of several agent-oriented software methodologies which claim to support the construction of open systems.

However, even though agent-oriented software methodologies exist to support the development of open systems, they are lacking when dealing with the incorporation of new components (or agents) to an existing system. In particular, these methodologies do not address two different but very related problems:

- how to specify the facilities provided by the existing system for those interested in the development of new agents; and
- how to design and construct mechanisms to ensure that the integrity of the system is not violated at run-time by new agents.

Solving these problems requires the accomplishment of some non-trivial tasks. In order to solve the first problem of specifying the facilities provided by the system, we must first accomplish the selection of appropriate abstractions on which to base the specification. For the second problem of ensuring that the integrity of the system is not violated at run-time, mechanisms for monitoring the behaviour of the system and evaluation of its characteristics must be provided [57].

Although complete solutions to these problems are highly application and platform dependent, we can, nevertheless, separate more general problems from more specific ones and provide partial solutions. In particular, in order to create agents that are eventually incorporated into an existing system, developers need to know what facilities are provided by the system, and the way in which they can access them, so that they can design new agents in accordance with these characteristics. In addition, developers must be aware of the rules of behaviour of the system, and design new agents in such a way that those rules are observed at run-time. From the viewpoint of maintaining the integrity of the system, this is particularly important in the case of multi-agent systems, because the autonomy and pro-activity exhibited by agents can easily lead to unexpected behaviour.

In this chapter we present a model for the specification of open multi-agent systems based on organisational concepts, and then apply it to create a mechanism for checking that a specification is observed at run-time. With this in mind, we divide the structure of the chapter in the following way. In Section 7.2 we analyse the characteristics of a specification in open multi-agent systems, that is, *what* must be included, and *how* to express it. Then, in Section 7.3 we formalize such a specification. After that, we focus on the problem of how to check that such a specification is observed at run-time in Section 7.4, which includes the design of a mechanism that monitors the observance of a specification. Finally, Section 7.5 contains the conclusions of the chapter and indicates future work.

7.2 Basic concepts

7.2.1 A layered model for open multi-agent systems

A specification describes the components of a system and their composition. Additionally, in the case of multi-agent systems, apart from describing the components (agents) and their composition, it is also necessary to include their interaction. The reason for this is that, since agents are not passive service-providers but pro-active and autonomous entities, their interactions need to be explicitly stated. Open multi-agent systems are complex in structure and, as a consequence, so are their specification. For this reason, and because many diverse aspects are involved in specifying agents, their composition and their interaction, we find it useful to decompose the functionality of an open multi-agent system into several parts, or *layers*. (Such a perspective is commonly used in the specification of open infrastructures as, for example, in the X/Open platform and the ISO model of communications [65].) Each layer deals with a particular aspect and rely on the previous layer to fully specify the corresponding functionality. Furthermore, each layer should add a higher level of abstraction, and at the same time impose further constraints on the way agents behave.

Thus, an appropriate layered decomposition of open multi-agent systems must cover all their intrinsic characteristics, from network communication to organisations. Our proposal for decomposing open multi-agent systems is depicted in Figure 7.1, obtained by listing the different aspects of any multi-agent system by order of level of abstraction, as follows

- The lowest layer, *Communication Protocols* (CP), deals with the low level protocols for communication; for example, IIOP and HTTP. Although strictly this layer does not contain elements of agenthood, it specifies the distributed aspect of a multi-agent system at its most basic level.
- The next layer, *Agent Platform* (AP), relates to the infrastructure in which the agents operate, in terms of which agent services are provided and how. Such agent services include the management of agents in a system — registration and deletion — white and yellow pages services, and message routing.
- The *Agent Communication Language* (ACL) layer is concerned with the language employed by the agents to exchange messages during their discourse, particularly those elements of a language that are application independent. Among these elements, some of the most important are the *communicative acts* (or *performatives*), which denote an intention from the sender of the message for the receiver of the message to perform, such as to execute or cancel an action. Instances of standard agent communication languages are FIPA-ACL [35] and KQML [34].

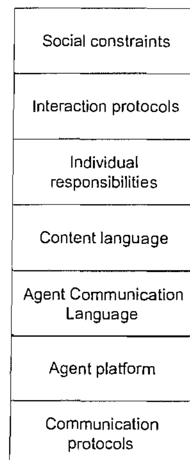

FIGURE 7.1: Layered decomposition of open multi-agent systems

- The *Content Language* (CL) layer deals with the language used to represent concepts specific to each application domain. Together, the content language and the agent communication language layers constitute the language in which the communication between objects occurs.

- The *Individual Responsibilities* (IR) layer refers to the functionality that each agent is capable of performing without interacting with other agents. In other words, this layer specifies what is expected from each agent as an individual entity.

- The *Interaction Protocols* (IP) layer relates to the interactions that agents perform to fulfil their goals. Instead of referring to details of individual messages, as is the case of the ACL and CL layers, this layer focuses on the way specific groups of messages are used to carry out tasks concerning the overall system.

- Finally, the *Social Constraints* (SC) layer establishes the expected social behaviour of the agents. The concept of social organisation is arguably what differentiates the multi-agent approach from other software approaches. Although some of the organisational elements of a system are specified in previous layers — the participating entities (IR layer), and their interactions (IP layer) — the social constraints complete the organisational structure by imposing rules on the way these elements are combined.

As can be observed, this decomposition is independent of any particular architecture, platform or methodology, and focuses on high-level aspects of the system, rather than on the detailed composition of the components.

It should be noted that this decomposition is similar to the Abstract Architecture[1] defined by FIPA with the purpose of promoting interoperability and reusability. Such an architecture considers that agent-based systems can be decomposed into two parts, a (domain-specific) *application* layer, and an *abstract architecture* layer, which involves *agent communication*, *agent management* and *agent message transport*. However, although the application layer might correspond to the IR, IP and SC layers, there is no explicit usage of organisational concepts. Similarly, the agent communication component corresponds to the CL and ACL layers, and the agent management and agent message transport correspond to the AP and CP layers, respectively, but in spite of this correspondence, the FIPA architecture dooes not exploit a layered structure in its components.

Since any multi-agent system consists of these layers, a complete specification of an open multi-agent system must take into account all the layers. However, our concern is not to provide a complete multi-agent system specification, but to address those aspects relevant to the construction of open systems, building on prior work and infrastructure. Thus we avoid the enormous effort that would be required, because of the variety of elements involved, and focus on creating specifications for the manageable subset of layers that have not adequately been studied before, and which address the focus of this thesis. More specifically, the lower layers of the model — the CP, AP, ACL and CL layers — have received much attention in the past, as is indicated by the existence of *de facto* standards (e.g. IIOP [59], FIPA [35], KQML [34], and KIF [4]). In contrast, from the viewpoint of *specification*, the three upper layers — IR, IP and SC layers — have been the less studied. Moreover, these three layers are specifically relevant to the organisational metaphor that we have used extensively through this thesis to model various aspects of multi-agent systems.

Based on this decomposition, we structure the specification of a system into parts, or *sections*, each corresponding to a layer. The content of the sections and the order in which they appear are the same as those of the layers in the decomposition. Regarding *how* to describe the contents of each section, the natural choice is to employ *standards*, since they are based on agreed terms and concepts, and provide a commonly accepted way to describe systems. For example, in the case of the ACL layer, the specification can make use of the FIPA agent communication language (FIPA-ACL) [35]. On the other hand, although at the IP layer some standards have emerged, further down the IR and SC layers lack not only standards, but also commonly agreed concepts. Thus, in order to progress with our goal of creating specifications, we need to adopt, or construct, appropriate conceptual abstractions which, on the one hand, can remove artificial distinctions between the different constraints of heterogeneous components (or agents) possibly developed by different groups, with different architectures and methodologies and, on the other hand, are still closely connected to real systems.

Although for the IP, IR, SC layers there are no *standard* abstractions, the concepts of *role*, *protocol* and *organisation* are increasingly being used in fields such as agent-oriented software engineering as first-order abstractions to model relevant aspects of open systems. In consequence,

[1] http://www.fipa.org/specs/fipa00001/SC00001L.html

we use these abstractions as a basis for open systems specification, and present below a brief review of the concepts (for more details, see Chapter 3).

Although the concepts of *role*, *protocol* and *organisation* can provide an appropriate means with which to specify the IP, IR and SC levels, employing them can be problematic. First, despite many characterisations and definitions of these concepts being available in the literature, none is entirely appropriate for our purpose. This is because we require a characterisation that abstracts *what* is needed from the components and their interaction, bypassing the details of *how* it is actually achieved, which is the usual stance taken. At the same time, it is desirable that such characterisations are similar to those used in existing agent-based methodologies in order to minimise incompatibilities when developing a new agent. Finally, the characterisations must be independent from any specific implementation. In the following subsections we present the characterisations for roles, protocols and organisations that will, in turn, be used to construct the specifications of our models.

7.2.2 Roles

A role represents a position in charge of performing a specific service in an organisation. Examples of roles in the Conference Management System (an hypothetical system for managing conferences, whose statement problem is included in Appendix A) are *Author* and *Reviewer*. In general, at run-time roles can be played by one or several agents, and a single agent can play one or several roles. Role is a concept that is intuitive, simple and easy to understand; it is used in different agent-oriented software methodologies, and serves as a basis for other useful abstractions, such as protocols and organisations.

Although the concept of role is extensively used in the agent literature, there is no a common characterisation. For the purposes of specifying an open system, we see a role as consisting of the following parts.

- Name: a unique identifier of the role.
- Description: brief text explaining the purpose of the role.
- Services: functionality that the role is expected to perform.
- Non-functional requirements: special conditions that an agent must satisfy at run-time to play this role, such as the minimum amount of a certain resource, or confidentiality.

7.2.3 Protocols

A role rarely accomplishes its objectives by itself, but interacts with other roles, as determined by a protocol, which is a high level representation of the interaction of roles to accomplish an objective. A protocol is a representation of pre-defined patterns of interaction, and is used not

only in the context of methodologies but also in the analysis of specific mechanisms, such as negotiation.

The concept of protocol is also commonly used in the agent literature, but its definition varies very from paper to paper. For example, in Gaia, protocols are characterised only by name, initiators, partners, inputs and outputs. Other approaches provide additional information, for example the messages exchanged (communicative act and content), their sequence, the tasks that they trigger, and even the state changes of the agents as a result of exchanged messages. For our purposes, we use the following characterisation, which has the benefit of considering all the key elements to describe an interaction between roles, but without compromising to any particular platform or implementation.

- Name: a unique name in the system. This serves as an identifier for the protocol.
- Initiator role: the role that begins the interaction. This role should not be repeated in the next item.
- Partner roles: other roles involved in the interaction. We will also refer to partner roles as *collaborators*.
- Inputs: information needed to perform the interaction. Inputs are environmental entities passed as parameters to the protocol and whose values are not affected.
- Outputs: information obtained or modified as the result of the interaction. Outputs are environmental entities passed as parameters to the protocol and whose values can be modified.
- Messages: the messages exchanged in the protocol. The description of these messages contains the sender, the receiver, a form of identification (such as a *communicative act*) and, optionally, the content.
- The message sequence: the order in which the messages must be exchanged during the protocol. To describe this order, some constructors are provided, such as sequence, concurrence, and conditionals.

An example of a protocol in the Conference Management System is shown in Figure 7.2, in which the name of the protocol is *CallAndSubmission*, the initiator is the *Caller* role, the partners are the *Author* and the *Collector* roles, the input is the *call for papers*, and the outputs are the *paper* and a *confirmation of receipt*. Also, the messages are *call-for-papers*, *submit-paper* and *confirm-number*, and occur in that order.

7.2.4 Organisational rules

A multi-agent system is not completely defined by just a set of autonomous roles and their interactions. In fact, a multi-agent system typically possesses overall goals, and its components (even

sd CALL AND SUBMISSION Protocol

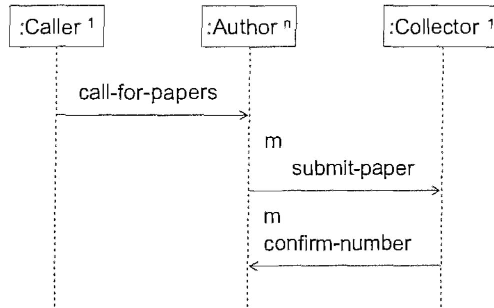

FIGURE 7.2: Example of a protocol specification

- *Enough reviewers must be allocated*
- *All papers must be reviewed*
- *An author cannot send the same version of the paper more than one time*
- *No agent can be both author and reviewer of the same paper*

FIGURE 7.3: Organisational rules for the Conference Management System

if self-interested) act in an orchestrated way in order to achieve them. However, this orchestrated behaviour does not emerge by itself — agents are pro-active and autonomous entities whose behaviour can become highly unpredictable in dynamic environments — but some mechanism is required to produce it. Organisations have proven to be an appropriate mechanism to fulfil this task. More specifically, organisational rules, which have been used in the design of multi-agent systems [134, 25], offer an appropriate analytical means to constrain the way in which the different elements of a system interact. If carefully designed, compliance with organisational rules at run-time can ensure the desired behaviour. To this end, we define organisational rules as constraints on the relationships between roles, protocols and resources, with the purpose of fulfilling the overall goals of the system. Some examples of organisational rules are included in Figure 7.3 for the Conference Management System.

We can classify organisational rules in different ways. One possible classification is based on the types of elements the rules constrain, giving place to *horizontal* and *vertical* rules. Horizontal rules are applied to elements of only one type. For example, a rule that applies to all the roles in the system is a horizontal rule, such as:

All the controller roles must maintain the same rate of operation.

On the other hand, vertical rules are those rules that apply to different types of elements. For example, rules that affect roles and protocols are vertical rules, as in

The selection process begins after all the papers have been reviewed three times.

Although this classification is useful in general discussion of organisational rules, from the viewpoint of checking their compliance, it is more convenient to divide organisational rules into *static* and *dynamic* rules. Static rules are those that need to be verified only at specific moments in the life cycle of the system. For example, the rules restricting the roles played by an agent can be verified simply at the moment the roles are assigned, as in the rule:

No agent can play the roles of buyer and seller at the sane time.

Dynamic rules, on the other hand, need to be checked frequently, or at moments that cannot be easily predicted. For example, rules restricting the use of resources or protocols need to be checked every time the resources are accessed or the protocols executed, and these events can occur at any time, for instance for the rule:

A paper must be reviewed exactly three times.

Most static organisational rules can be checked by means of a run-time component that keeps track of role assignments. Such a form of checking is simple, easy to implement, and most importantly, does not interfere with the normal operation of the system.

Dynamic rules are potentially more difficult to check than static rules, due to their diversity and dynamism. In principle, such checking can consist in monitoring every element of the system and evaluating the relevant rules. Roughly, this could be accomplished by obtaining the relevant information from the agents or, in the case of non-agentified elements — such as resources — from tailor-made modules that control their access. However, this procedure must be carefully designed not to significantly affect the performance and robustness of the system. Distinguishing these two types of organisational rules is helpful when devising procedures to monitor them at run-time, as can be seen in Section 7.4.

7.3 Specification of open multi-agent systems

We now use the basic concepts presented above for creating specifications of existing systems, in such a way that potential participants can determine the requirements and benefits of joining the system. In our case the targets of such a description are the designers of the agents.

Such a specification must be as neutral as possible, since the agents might be developed with varied techniques. However, at least some basic assumptions must be made; in particular, the use of some common, appropriate concepts is required. As was stated in Section 7.2, we use *role*, *protocol* and *organisation* as the basic concepts on which a specification can be constructed. In general, these abstractions appropriately model the characteristics found in multi-agent applications, namely agents, interactions and rules. In particular, they give rise to a set of models

that provide the documentation necessary both for developers and for automatic compliance monitoring in order for agents to join open systems in effective and managed ways.

We divide the specification into two parts. The first part deals with characterising and documenting the generic elements of the system that are always to be found, regardless of the nature of the system, the *general concepts model*. The second part comprises the agent-specific models based on organisational concepts, the *participants model*, the *interactions model* and the *social constraints model*. All of these models are presented below.

7.3.1 General concepts model

The general concepts model contains the description of the resources and entities of the environment that are necessary, in turn, in the description of the other parts of the specification such as protocols, activities and social constraints. Since these general concepts involve only information, we use a characterisation based on registers and fields. In other words, each concept is described by the elements it encompasses, and each of these elements is in turn described by its sub-elements. This process continues until the sub-elements are *non-decomposable* data structures such as strings, numbers or dates. To completely define the part-whole relationship between a concept and its elements, *cardinalities* are used, denoting how many elements can be present in a concept; for example, the concept *Paper* has one or more *author* elements. Figure 7.4 shows the general form of this model. The cardinalities are enclosed by square brackets to denote that they are optional. If not specified, a cardinality of 1 is assumed.

As an example of a fragment of a general concepts model, we consider the *Paper* concept of the Conference Management System. This concept consists of five parts: a title, an abstract, a body, the authors and their affiliation. The model for this concept is shown in Figure 7.5. In this figure, the + cardinality represents one or more parts, and the data structure *string* is used to denote short textual information, whereas *text* is used for potentially large text. Equivalently, this *Paper* concept is also depicted in Figure 7.6 by means of a UML class-type diagram, in which the boxes represent concepts and elements, the lines ending in diamonds represent the whole-part relationship, and the numbers near the lines denote the cardinalities (Entity-Relationship diagrams could also be used to this end, although we use UML for consistency).

7.3.2 Participants model

The participants model contains the description of each agent of the system, referring only to those individual characteristics that do not involve interaction with other agents, and that are independent of how the agent is implemented. Since we model agents by means of roles then, according to the characterisation of roles we employ, the participant model consists of the set of roles in the system and, for each of them, a list of their services and non-functional requirements.

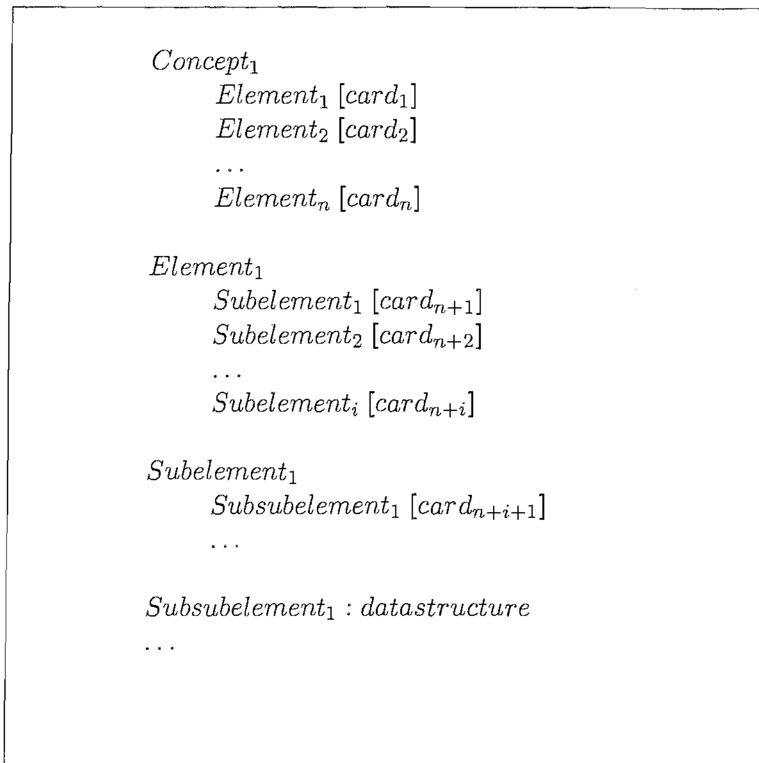

FIGURE 7.4: The general form of the General Concepts Model

Services are tasks that a role can perform without interacting with other roles. We propose a simple characterisation of services consisting of a name, the role that performs it, their input and output parameters, and a description of the task itself. Since the actual implementation of the process is not restricted by the specification, its description can be text, pseudocode or any formal description. Regarding the non-functional requirements, we follow a simple approach consisting of representing each requirement by an identifier-value pair, for example $(memory, 40)$, where the identifiers and their possible values have previously been defined.

The general form of the participants model is shown in Figure 7.7, in which requirements identifiers are denoted by id_i and their corresponding value by $value_i$. The square brackets indicate that the use of non-functional requirements is optional. As an example, Figure 7.8 presents a fragment of the participants model corresponding to the Conference Management System. This simple example shows three participants, each one having a service.

7.3.3 Interactions model

The interactions model describes the way roles interact by means of protocols. Our protocol characterisation is inspired by a simplified version of *sequence diagrams* similar to those of AUML [98], and represents the participating roles in the protocol, the messages they exchange, and the sequence of those messages. The messages are labelled with their communicative act

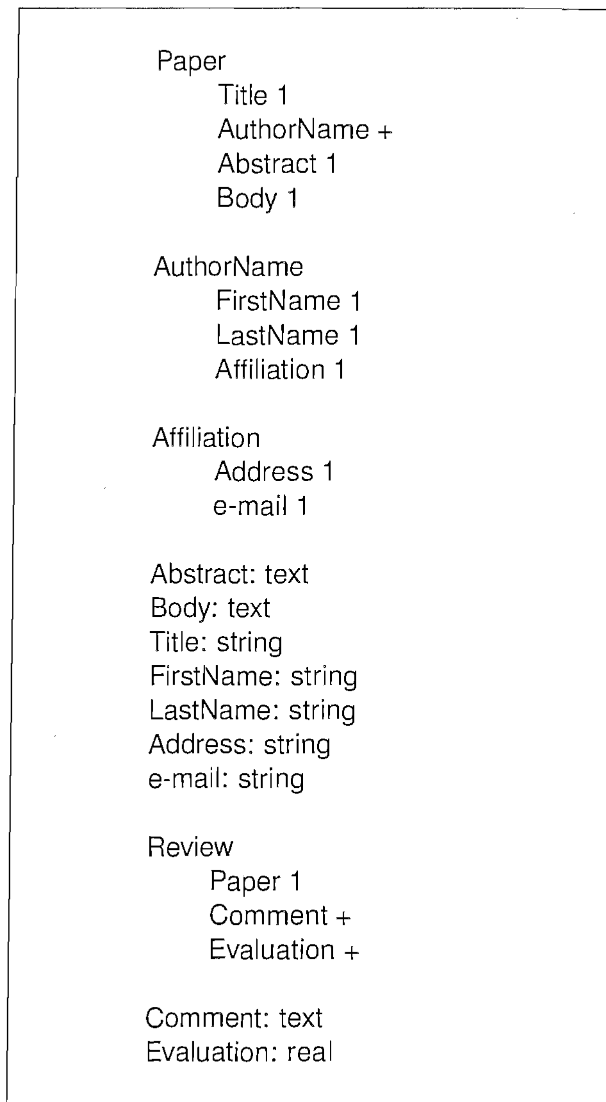

FIGURE 7.5: The application of the General Concepts Model to the CMS example

and content, or with an identifier (whose communicative act and content are defined elsewhere, e.g. in [35]). The communicative acts must be described in the agent communication language specified in the Agent communication language layer. In the same way, the content must belong to the content language specified in the Content language layer and the specification of general concepts.

As can be observed in Figure 7.9, the interactions model is composed of the list of protocols in the system. Each protocol consists of a list of participants (the first of which is the initiator of the protocol), a list of parameters, and a sequence of messages. The messages are formed of a sender, a receiver, a communicative act and a content (although, for simplicity, this is not shown in the figure). An example showing a fragment of the interactions model for the Conference Management System is presented in Figure 7.10, which contains two protocols, *SubmitPaper*

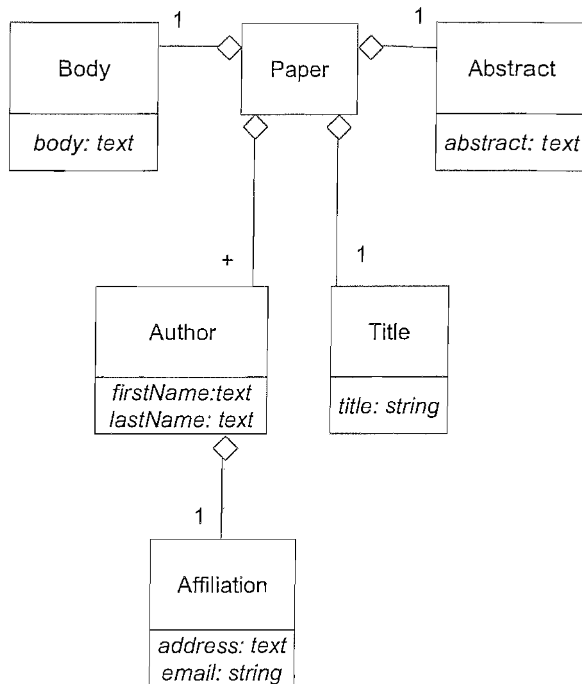

FIGURE 7.6: The paper concept

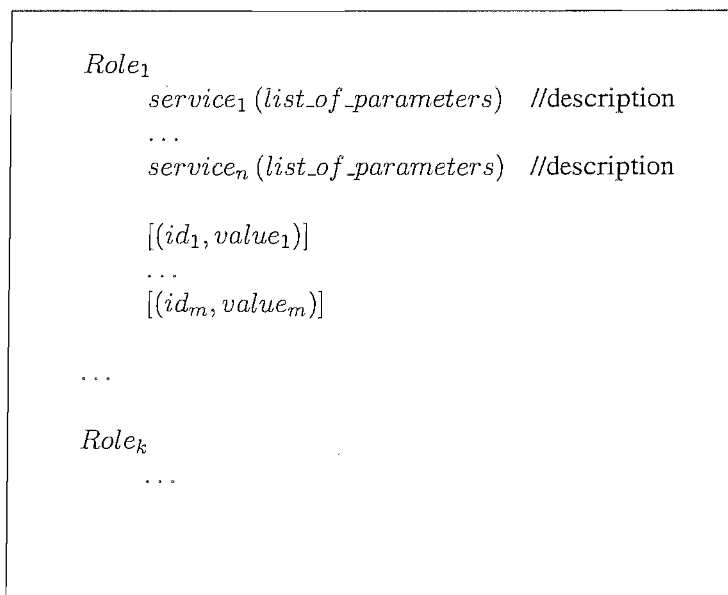

FIGURE 7.7: The general form of the participants model

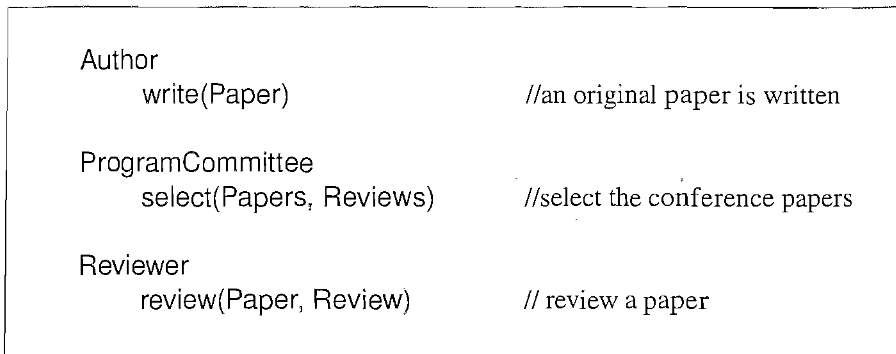

FIGURE 7.8: The application of the Participants Model to the CMS example

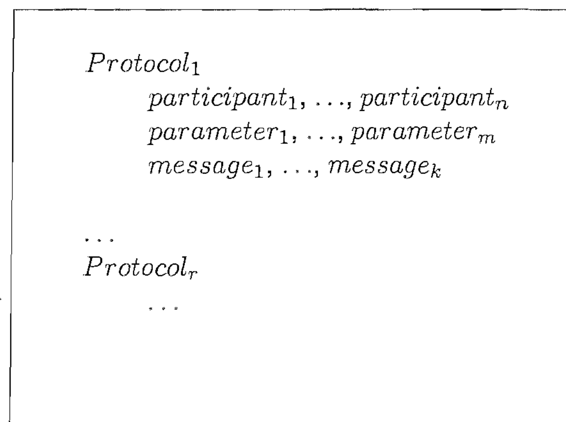

FIGURE 7.9: The general form of the interactions model

and *ReviewPaper*. The latter, for instance, employs two messages for carrying out the interaction between roles *ProgramCommittee* and *Reviewer*.

7.3.4 Social constraints model

The specification of social constraints contains the restrictions imposed on the agents' social behaviour. Such restrictions are represented by means of organisational concepts, more specifically, by organisational rules. Organisational rules are key to the definition of the organisation and thus of the system itself. For this reason, an agent attempting to join an existing system must be provided with the set of rules it must adhere to. The specification of social constraints is formed from the list of organisational rules of the system, expressed in the language defined in Section 3.3.4. The general form of this model is represented in Figure 7.11. An example consisting of two rules in the Conference Management System is shown in Figure 7.12, in which the first rule states that there must be at least five reviewers, while the second rule states that a paper must not be assigned to the same reviewer more than once.

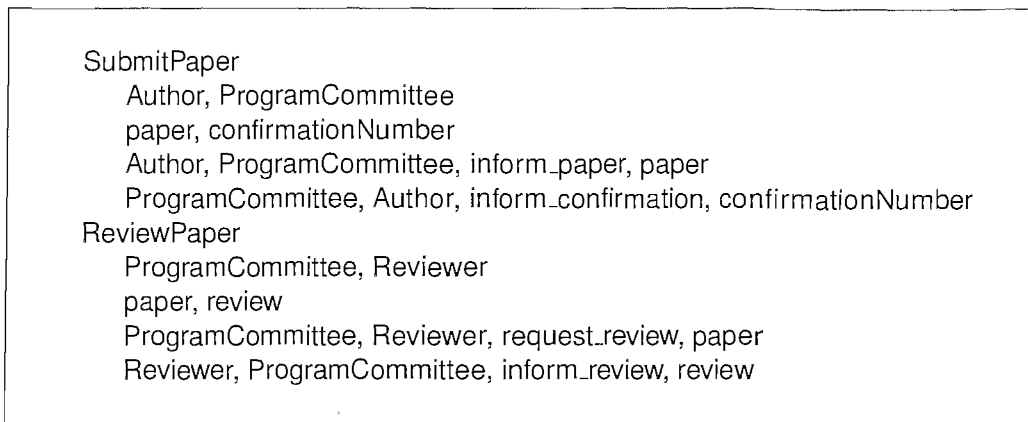

FIGURE 7.10: The application of the Interactions Model to the CMS example

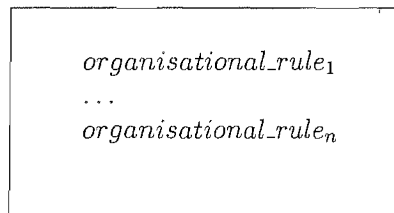

FIGURE 7.11: The general form of the social constraints model

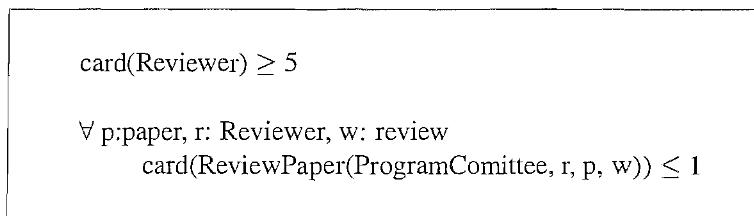

FIGURE 7.12: The application of the Social Constraints Model to the CMS example

7.3.5 Summary

We have presented a specification for open multi-agent systems. The specification consists of three main models and an auxiliary model, the latter being the general concepts model, and the former the participants, the interactions and the social constraints models. As illustrated in Figure 7.13, each of the main models corresponds to a layer in the system decomposition presented previously. The auxiliary model contains descriptions that complete the main models, as depicted in the figure by means of arrows. Also, we based the description of the participants, interactions and the social constraints models on well known abstractions of roles, protocols and organisational rules, respectively.

Up to this point, in this chapter we have focused on the creation of a system specification. Based on the results obtained here, in the following sections we explore the problem of ensuring that

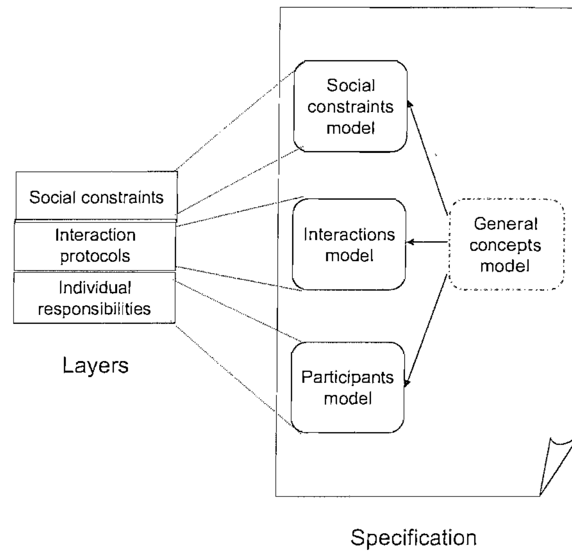

FIGURE 7.13: Description of the UpdateCall service

what is stated in the specification is observed at run time. Roughly, our approach consists of checking that the actions performed by an agent do not violate any of conditions stated in the sections of the specification. However, before proceeding, we present a formal model of the specification and consider the problem of examining that the specification is complete and free of inconsistencies.

7.3.6 A model of open systems

In an open multi-agent system specification the details of the internal structure of the agents are not important, but only their externally visible functionality. This is because the agents in the system may be constructed by different developers and following different techniques. For the same reason, the implementation details of the protocols are not relevant, but only their patterns of interaction. This ensures that the agents need not to be developed with the same tool, should they comply with the rules of the system. In this section we present a formal model for open multi-agent systems, based on organisational concepts, and that abstracts the functionality of the agents and the way they interact, regardless of implementation issues.

The system

We define a model for an open multi-agent system as a tuple $\langle \mathcal{E}, \mathcal{N}, \mathcal{P}, \mathcal{S}, \mathcal{O} \rangle$, where:

1. $\mathcal{E}$ is a 4-tuple of set of elements of the system;

2. $\mathcal{N}$ is the set of the *roles' non-functional requirements*;
3. $\mathcal{P}$ is the set of *protocols*;
4. $\mathcal{S}$ is the set of *services*; and
5. $\mathcal{O}$ is the set of *social constraints*.

The identifiers

$\mathcal{E}$, the tuple of elements in the system, has the form $\langle R, P, S, D \rangle$, where each entry is a set whose elements are identifiers, as follows:

1. R is the set of role identifiers;
2. P is the set of protocol identifiers;
3. S is the set of service identifiers; and
4. D is the set of general concept identifiers;

The non-functional role requirements

The elements of the set $\mathcal{N}$ have the form (r, n, v), where $r \in R$, n denotes a type of non-functional requirement, and v represents a possible *value* of n. The interpretation of this is that such a role requires at least that value for the non-functional requirement in order to be played. For example, in the conference management system,

$$(\mathit{ProgrammeCommitteeChair}, \mathit{confidentiality}, 1)$$

indicates that the role *Chair* must comply with the highest (1) confidentiality. However, it must be noted that the list of non-functional requirements and their associated values are highly dependent on the application and platform used.

The protocols

Each element of $\mathcal{P}$, the set of protocols, is a 5-tuple of the form (p, i, C, A, M), where:

1. $p \in P$ is a unique protocol name,
2. $i \in R$ is the initiator of the protocol,

3. $C \subset R$ is the set of collaborators, that is, the roles that participate in the protocol, apart from the initiator,

4. $A \subset D$ is the set of input and output parameters,

5. M is the allowed sequence of messages, expressing the order the messages must follow during the execution of the protocol. This is a sequence of instructions, each of which is either a message or a *compound message*. A compound message encompasses a *connector* and a *set* of messages, and represents the concurrency connectors of AUML. Concurrency connectors are used as a means to express that multiple messages are sent at the same time, and are of three types: *and* (AND), *inclusive or* (OR), and *exclusive or* (XOR). In the first case all the messages are sent in parallel, while in the second zero or more messages are sent and in the last case only one message is sent. Each element of M, the set of messages of a protocol, has the form
   (r_s, r_r, b), where:
   $r_s \in R$ is the sender;
   $r_r \in R$ is the receiver; and
   b is the body of the message.

The services

S, the set of services, consists of elements of the form (r, n, B), where:
$r \in R$ is the role to which the service belongs,
$s \in S$ is a unique service name, and
$B \subset D$ is the list of parameters of the service.

$\mathcal{O} \subset \mathcal{L}$, the set of social constraints, contains the expressions that govern the function of the system. Each element in this set is an element of the language defined in Section 3.3.4.

Table 7.1 summarises this notation. For simplicity, we do not include the part corresponding to the sequence of messages, but only the structure of each message.

7.3.7 Ensuring information consistency

A specification describes a system from different perspectives; for example the specification of protocols deals with the interaction aspects while the specification of participants focuses on the individual aspect of roles. However, it is essential that these perspectives are not in contradiction, but describe the system in a *consistent* form. For instance, an organisational rule cannot reference a protocol that has not been defined in the specification of interaction protocols. For this reason, we need a mechanism for checking consistency in the specification. Such a mechanism can be implemented in different ways; for example, by means of a software tool the consistency can be checked every time the specification is updated. Whatever mechanism used, the following conditions must be checked.

$\mathcal{E}$	element identifiers		
	R	role identifiers	
	P	protocol identifiers	
	S	service identifiers	
	D	concept identifiers	
$\mathcal{N}$	non-functional reqs.		
	r	role to which applies	
	n	non-functional reqs. identifier	
	v	value	
$\mathcal{P}$	protocols		
	p	protocol identifier	
	i	initiator	
	C	collaborators	
	A	protocol parameters	
	M	sequence of messages	
		For each message:	
		s_e	sender
		s_r	receiver
		b	body
$\mathcal{S}$	services		
	s	service identifier	
	r	role	
	B	service parameters	
$\mathcal{O}$	social constraints		

TABLE 7.1: Summary of notation

1. The name of roles, protocols, responsibilities and general concepts must be unique.

2. All the protocols mentioned in the specification must be described in the specification of interaction protocols.

3. All the roles mentioned in the specification participate in at least one protocol and have at least one responsibility.

4. All the resources mentioned in the specification must be defined in the specification of general concepts.

As a final observation about achieving completeness and consistency in a specification, we should mention the use of organisational patterns. As discussed in Chapter 3, organisational patterns represent the organisational structure of a system, including the roles of the system, their protocols of interaction, and organisational rules. Organisational patterns may serve as a basis on which the specification of a system can be completed, because they already contain much of the relevant information. Additionally, since such patterns can be regarded as free from incomplete or inconsistent information (assuming they have been used with success in several applications), the amount of information needed to be checked is reduced. The usefulness of

this approach would be improved by means of a software tool to support the creation of a specification by *importing* the appropriate organisational pattern, editing of the specification and checking completeness and consistency.

7.4 Compliance monitoring

As mentioned previously, our approach to the problem of ensuring the integrity of an open system is to monitor, at run-time, that the system acts in compliance with the specification. In other words, we are assuming that the integrity of a system is ensured if all the conditions expressed in the specification are observed. Checking compliance with the specification in open multi-agent systems must be done at run-time, because, by definition, there is no control over how the different components of the system are designed and developed. An additional benefit of monitoring this compliance is that it guarantees that the system behaves correctly, since organisational rules are part of the specification, and organisational rules ensure the correct behaviour of the system.

A different approach towards ensuring the correct behaviour of agents in an open system is described in [27], in which Dignum et al. present a norm-based organisational model, OMNI, that considers the use of *violations*, *sanctions* and *enforcing roles* to enforce compliance of norms. A violation is a condition that indicates that an agent is in an illegal state, sanctions are the actions carried against the violator, and enforcing roles are the roles in charge of detecting the violation. OMNI provides procedures for obtaining these elements from *norms*, which are situated at a more abstract level. Compared to our approach, the violations considered in OMNI are less general than the organisational rules we employ, in terms of the type of constraints that can be described. In addition, our approach is situated at a level closer to implementation than OMNI.

Monitoring the compliance of the specification involves the analysis of a large number of conditions and situations caused by the actions of agents, for example executing a protocol or performing a service. We find it useful to divide this monitoring according to the *nature* of the action that caused the situation. Thus, we classify the actions of an agent into *static* ones and *dynamic* ones. Static actions are those that occur at one specific moment of the agent life cycle, typically during the entry of the agent to the system, or when a role is assigned to the agent. Dynamic actions, on the other hand, are those which can occur at any other moment. In accordance to this, we divide monitoring into static analysis, and dynamic analysis, both of which are presented below.

7.4.1 Static analysis on agent entry

The moment of its entrance to a system and every moment it attempts to play a role are critical points in an agent life cycle, since it is important to ensure that the agent is suitable for the system or the role. In order to obtain meaningful results, we have to make some assumptions

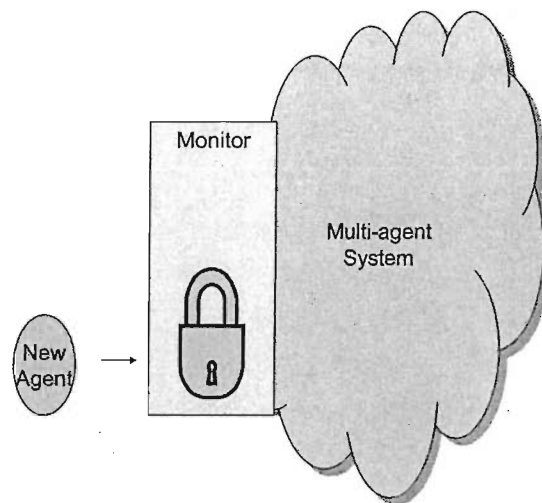

FIGURE 7.14: The function of the monitor

regarding the way agents enter a system. We assume that each time an agent attempts to enter the system, some mechanism is used to decide whether its entry is accepted. Once accepted, agents can play roles, or quit playing roles. Both actions are notified to the system, and the former needs to be authorised.

To clarify this point, suppose that an agent intends to enter the system. It must first receive approval from a run-time component, hereafter called the *monitor*, as depicted in Figure 7.14. As suggested in the figure, the only way for an agent to access the system is by getting approval from the monitor, based on the *characteristics* of the agent and the specification of the system. The monitor provides a means to consider aspects that are verifiable statically, for example to detect if a protocol has an incorrect *initiator role*, in the sense that it does not match what is stated in the specification. However, the monitor does not consider aspects that are not verifiable statically, such as if a protocol is executed at the wrong moment.

With these considerations in mind, we proceed to analyse how to check the observance of a specification. We do this model by model, but omitting the general concepts model, since there is nothing to analyse because it only supports the other models.

7.4.2 Run-time participants analysis

The run-time analysis for the participants has the aim of ensuring that the agents comply with the participants' model of the specification. This can be done statically, at the moment the agent requests authorisation to play a given role. Note that the agent can be playing other roles, or no role at all, before attempting to play a specific role. When an agent requests authorisation to play a role, the monitor must check that the characteristics of the agent, and the way it implements the role, match the conditions stated in the participants model. More specifically, given the role

in question, the services as implemented by the agent, and the resources that the agent possesses, the monitor must check that the following conditions hold.

- The role that the agent intends to play exists and is available; that is, the role has not exceeded its cardinality.
- The agent has enough resources to satisfy each of the non-functional requirements specified in the participants model.
- The agent implements all the services specified in the model, in the way they are specified, in terms of name and parameters.
- Optionally, for more strict checking, the agent does not implement other services apart from those specified in the model.

Note, however, that checking the services in this way only offers a guarantee that their *interfaces* have been correctly implemented, but does not say anything about whether they have been correctly implemented; for example, if instead of adding two numbers, they are multiplied.

7.4.3 Run-time protocol analysis

During the entry of an agent to the system, we can also check, to some extent, whether the protocols implemented by the agent correspond to those specified. Essentially, the procedure is a matter of matching the characteristics of both protocols: those of the agent implementation and those specified in the system. Most of the checking is straightforward, except the part regarding the sequence of messages of the protocol, which depends on how many features of the sequence diagrams are considered. According to this, the algorithm is divided into two parts: matching the head (which deals with matching the initiator, collaborators and parameters) and matching the messages (which deals with checking the sequence of messages). Protocols are accepted only if they are accepted in both parts. However, it must be noted that this procedure does not check the dynamic characteristics of the protocol, such as the *actual* sequence in which the messages are sent, nor the actual content of the messages.

In the following, such a procedure, together with its inputs and outputs, is presented, and have been illustrated with diagrams, Table 7.2 shows the inputs in tabular form, while Figure 7.15 shows an example of a sequence of messages of a hypothetical protocol.

7.4.3.1 Algorithm for matching the head

Matching the head involves checking that the role exists and that the protocols correspond to those specified in the interactions model. The interactions model was presented in Section 7.3.3, and is refined below using a notation that is more appropriate for expressing the algorithm.

	Name of role intending to play
	List of protocols intending to use
For each protocol:	**Initiator**
	Collaborators
	List of Input parameters
	List of Output parameters
	Sequence of messages
For each input and output parameter:	**Name**
	Type
For each message:	**Sender**
	Receiver
	Communicative act

TABLE 7.2: Inputs for the protocol checking procedures

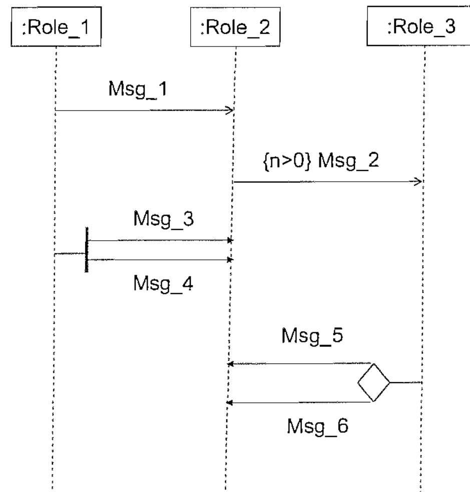

FIGURE 7.15: Different Connectors in AUML

Let $R = \{r_1, r_2, \ldots, r_k\}$ be the set of roles of the system (where k is the number of roles), and Q_l the set of protocols associated to role r_l.

Q_l contains the protocols associated with role r_l, so it can be written as $Q_l = \{q_1^l, q_2^l, \ldots, q_{m_l}^l\}$, where m_l is the number of protocols associated with role l, and each q_j^l denotes a protocol and thus have the form

$q_j^l = \left(p_j^l, i_j^l, C_j^l, A_j^l, M_j^l\right)$, where:

p_j^l is the name of the protocol,

$i_j^l \in R$ denotes the initiator,

$C_j^l \subset R$ denotes the collaborators,

A_j^l is the (ordered) sequence of parameters of the protocol, each consisting of a *name* and a

INPUTS:

r the role in question; and
$Q \subseteq \mathcal{P}$, the set of protocols involving r, as implemented by the agent

OUTPUT:

acceptance:
true if the header of the protocol complies with the specification;
false otherwise

ALGORITHM:

acceptance = $false$
$r \notin R \Rightarrow$ **exit**
$\exists e$ such that $(r = r_e) \wedge (1 \leq e \leq m)$
$\forall (p, i, C, A, M) \in Q_r$
  $p \notin \{q_1^e, q_2^e, \ldots, q_{m_e}^e\} \Rightarrow$ **exit**
  $\exists t$ such that $(p = q_t^e) \wedge (1 \leq t \leq m_e)$
  $i \neq i_t^e \Rightarrow$ **exit**
  $C \neq C_t^e \Rightarrow$ **exit**
  $\forall (a, y) \in A$
    $(a', y') = nextElement\,[M_t^e]$
    $(a' \neq a) \vee (y' \neq y) \Rightarrow$ **exit**
acceptance = $true$

FIGURE 7.16: **Algorithm:** MATCHING THE HEAD

type, so we can express it as
$A_j^l = \left\langle (a_1, t_1), (a_2, t_2), \ldots, \left(a_{m_j^l}, t_{m_j^l}\right) \right\rangle$, where m_j^l is the number of parameters of the protocol, and finally
M_j^l is the sequence of messages.

The algorithm is presented in Figure 7.16 and, as can be observed, is straightforward and consists of checking the compliance of the protocol name, the initiator, the collaborators and the sequence of parameters of the protocol.

7.4.3.2 Algorithm for matching the messages

In the second part of the procedure, matching the messages, the objective is to check that the sequence of messages stated in the specification is equivalent to the sequence of messages implemented by the agent, so that any possible difference in the expression of the protocol is not important for execution. (From this perspective, we can ignore several features of sequence diagrams, but we do have to consider some others which are relevant when describing a sequence of messages.)

Before proceeding with the algorithm, it is worth mentioning the extent of the algorithm in terms of how the sequence of messages is formed. Our representation of protocols is based on AUML sequence diagrams [66], which are rich in features, some inherited from UML [43] sequence diagrams and some exclusive to agents. Included in these features are multiplicity of the messages — that the number of messages sent and the number of receivers of the messages must correspond to those of the specification — and the type of message delivery — synchronous or asynchronous. Also included are two types of message structure: conditions and concurrency connectors. A condition is a logical expression that determines if a message is sent or not. As was mentioned before, concurrency connectors are used as a means to express that multiple messages are sent at the same time and are of three types: *and* (AND), *inclusive or* (OR), and *exclusive or* (XOR).

However, for our purpose (checking whether two sequence diagrams represent essentially the same protocol) not all the features are relevant. While we need to consider the roles involved in the protocol and their existence in the system, and the *and*, *or* and *exclusive or* parallel connectors, the conditions of messages can be ignored since they are meaningful only at the execution of the protocol. We also left unconsidered: agents, since we only allow roles as participants of protocols; lifelines and threads of interaction, since they are not relevant in the functionality of the protocol; nested and interleaved protocols, since they are not considered in our definition of protocol; and protocol templates, for the same reason.

Since this algorithm is meant to be executed statically, it simply checks that the sequence of messages of a protocol matches the sequence specified in the system, but in the case of messages joined by a concurrency connector, the messages can appear in any order. Conditions are just ignored as they are relevant only at run-time.

To describe this algorithm we make use of the following functions. The first two functions operate on a message instruction, while the last two operate on a compound message. The $message$ function returns $true$ if the message instruction is a simple message, and not joined to other messages by a concurrency connector. The $compound_message$ function returns $true$ if the message instruction is a compound message, (a set of messages joined by a concurrency connector). The $connector_of$ function denotes the concurrent operator of a compound message (an element of $\{AND, OR, XOR\}$). Finally, the $set_of_messages$ function returns the set of messages of a compound message. Note that this function returns a *set*, not a sequence, since the order of the messages is not important.

The algorithm is presented in Figure 7.17. As can be observed, for the protocol to be accepted, the messages are compared. Simple messages are examined for equality, whereas for compound messages of type OR and XOR, equality is not required, but being a subset is enough. An example of a sequence of messages that does not match the sequence diagram of Figure 7.15 is shown in Figure 7.18, in which the compound message in the second position of the sequence is not a subset of the corresponding compound message of the diagram.

INPUTS:

$S = \langle m_1, m_2, \ldots, m_n \rangle$, the sequence of message instructions as described in the specification
$S' = \langle m'_1, m'_2, \ldots, m'_n \rangle$, the sequence of message instructions as implemented by the agent

OUTPUT:

acceptance

ALGORITHM:

acceptance = **false**
$\forall l \in \{1, \ldots, n\}$
  $message\,(m_l) \Rightarrow$
    $m_l \neq m'_l \Rightarrow$ **exit**
  $compound_message\,(m_l) \Rightarrow$
    $connector_of\,(m_l) = \mathbf{AND} \wedge$
      $set_of_messages\,(m'_l) \neq set_of_messages\,(m_l) \Rightarrow$ **exit**
    $connector_of\,(m_l) \in \{\mathbf{OR}, \mathbf{XOR}\} \wedge$
      $\neg\,(set_of_messages\,(m'_l) \subseteq set_of_messages\,(m_l)) \Rightarrow$ **exit**
acceptance = **true**

FIGURE 7.17: **Algorithm:** MATCHING THE MESSAGES

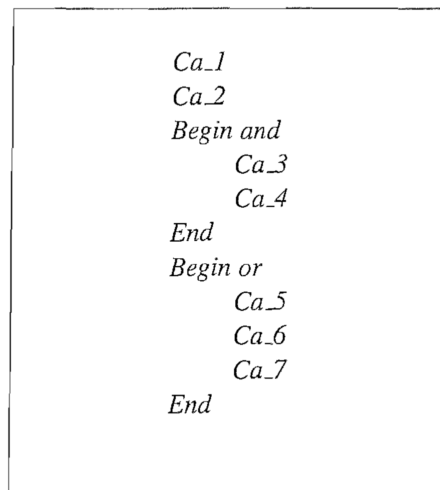

FIGURE 7.18: Example of sequence of messages

As can be observed, the algorithm for matching the messages presented above checks exact matches. However, under certain circumstances (for example under tight constraints of efficiency), a more relaxed form of checking might be convenient, or enough. In those cases, instead of checking that two protocols are identical, it can be checked whether one protocol is a particular instance of the other. For example, regarding the OR concurrency connector, instead of checking that both protocols contain the same messages, it might be enough to be sure that the messages of one protocol are a *subset* of the messages of the other protocol.

7.4.4 Run-time organisational rules analysis

As stated in Section 7.2.4, organisational rules are classified into static and dynamic rules, static ones being those that apply only at the moment at which an agent enters the system. In general, among all the possible types of organisational rules, just those dealing only with roles can be checked statically (which leaves out rules dealing with protocols and resources). The most common kinds of this type of rule are the following.

- Cardinality of roles. These rules establish the maximum number of times a role can be played at the same time; for example, *no more than 20 reviewers are allowed.*
- Sequence of roles. These rules constrain the sequence in which the roles can be played; for example, *an agent cannot play the role of buyer if it has not played the role of employer previously.*
- Exclusive roles. These rules express the fact that two or more roles cannot be played by the same agent at the same time; for example, *an agent cannot play the role of buyer and seller at the same time.*

In Figure 7.19 an algorithm is presented to check the compliance of static organisational rules. As can be noted, the algorithm is straightforward and consists of checking each of the different type of rules mentioned above.

In order to check these types of rules, a record must be maintained of the active agents in the system, together with the roles they are playing and the roles they have previously played. Figure 7.20 depicts a data structure that fulfils this purpose, in which a list of the active agents is maintained. Any active agent, for example A_i, has two lists associated with it, one for the role currently being played, and one for the roles played in the past. The algorithm assumes that the data structure accurately keeps track of all the entrances and departures of the agents to the system.

```
INPUTS:
    The list of organisational rules involving roles
    The agent
    The role attempting to be played

OUTPUT:
    acceptance

ALGORITHM:
acceptance = false
Select those rules involving the input role
Evaluate each of these rules
if the rule is about cardinality then
    if according to the register, the role has reached its maximum cardinality then
        acceptance = "exceeds maximum cardinality"
        exit
    end if
else if the rule is about sequence then
    if the rule states that some other role must be played before and,
        according to the register, that is not the case then
        acceptance = "violates sequence"
        exit
    end if
else if the rule is about exclusiveness then
    if according to the register, the offending role is being played then
        acceptance = "violates exclusiveness"
        exit
    end if
end if
```

FIGURE 7.19: **Algorithm:** CHECKING STATIC ORGANISATIONAL RULES

7.4.5 A design for checking static conditions

In the previous sections, we focused on describing the information and procedures required for checking the observance of static conditions. Based on this, in the following we present a high level design that shows how this checking can be carried out.

For the purpose of checking, the agent must provide the following information to the monitor.

- The roles it is intending to play.
- The protocols in which it will participate, either as the initiator or as a collaborator.
- The list of services it will perform.

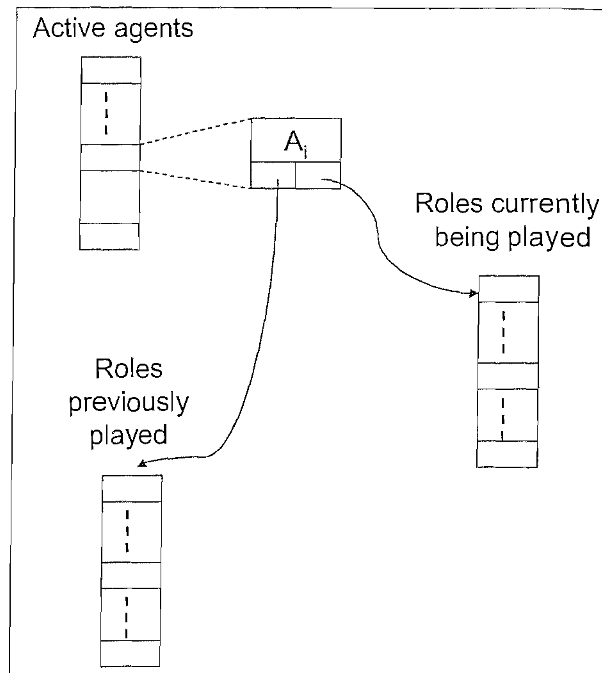

FIGURE 7.20: Data structure of agents and their roles

This information is represented in Figure 7.21, surrounded by an oval. The figure also shows other relevant information required by the monitor, which consists of the specification of the system (surrounded by bins), and the register with the role assignments (surrounded by a cloud). Figure 7.22 shows the main entities stored in the monitor's database. As can be seen, the system maintains a list of roles where each role performs one or more protocols. Protocols are formed of initiator, collaborators, input and output parameters and a sequence of messages (for a description of these characteristics see Subsection 7.2.3). Each message is described by the role that sends it, the role that receives it, the communicative act involved, and the position the message occupies in the sequence.

The monitor is formed of four well defined components, which are shown in Figure 7.23 and described below. The *Interface to agent information* module is in charge of obtaining the relevant information from each agent intending to join the system (see Section 7.3). In the simplest case, the module may obtain the information from a file. However, in some implementations the information could be obtained directly from the code of the agent. In the cases in which the specification is not actually tied to the code, there may be inconsistencies, the purpose of the monitor being to check compliance rather than ensuring integrity.

The *System specification* database contains the specification of all the roles in the system (see Section 7.3). The database is organised by roles and consists mainly of text fields. Its main job is to retrieve the information needed by the checker module, and occasionally to update the information of the system.

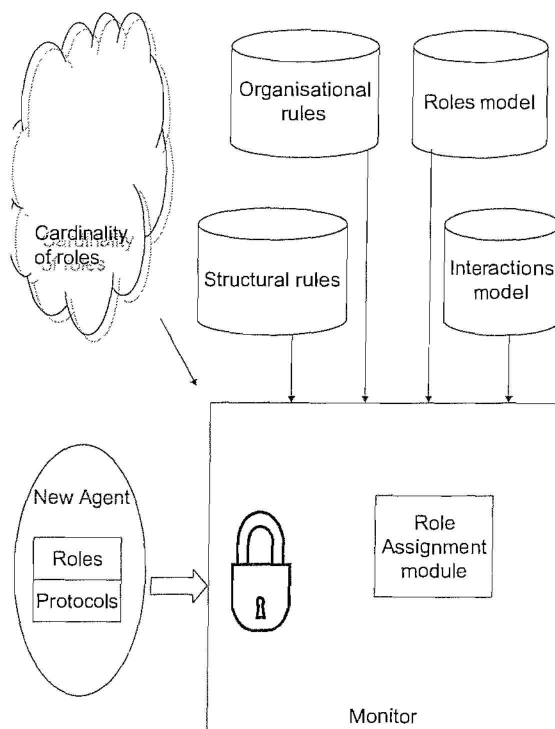

FIGURE 7.21: Information needed by the monitor

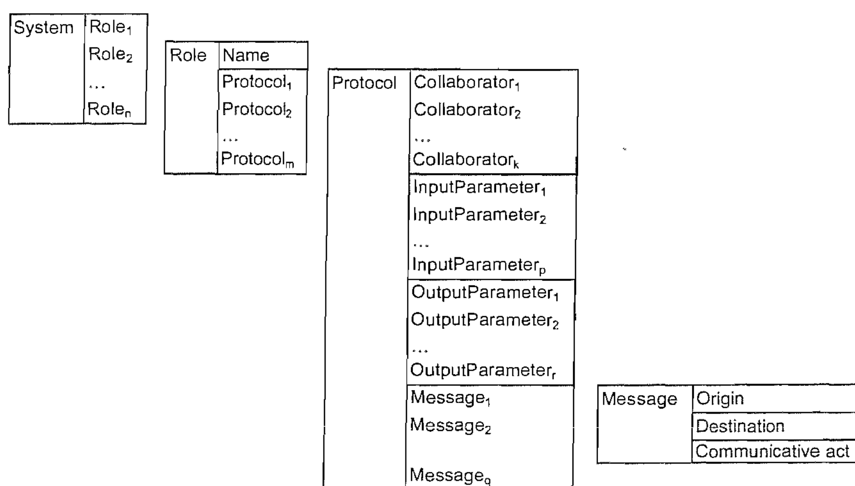

FIGURE 7.22: Monitor's database

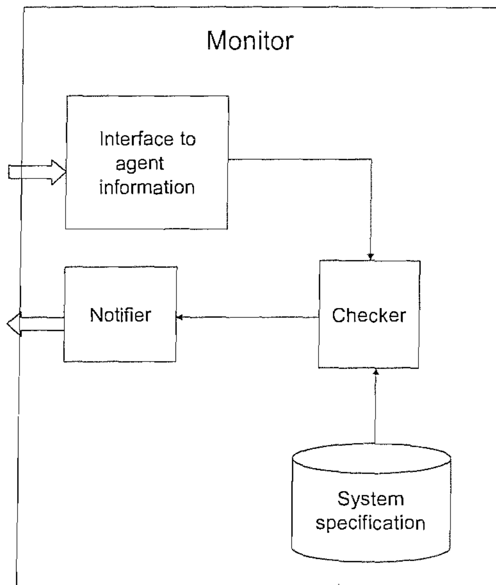

FIGURE 7.23: Components of the monitor

The procedures to decide if the agent complies with the system specification are contained in the *Checker* module. In the case of checking if a sequence of messages complies with the specification, the data flow is depicted in Figure 7.24. As can be observed, the problem of deciding if the two sequences of messages match is translated to the problem of deciding if two Finite State Machines (FSM) are *equivalent*, and this is a problem whose solution is well known in Computer Science [83]. The procedure to obtain a FSM from the sequence diagram is shown in Figure 7.25, which basically consists of separating the creation of states and transitions in accordance to the type of messages of the sequence diagram.

The last module, the *Notifier*, is in charge of notifying the acceptance or rejection of the agent to join the system. The actual implementation of this module depends on who is to be notified, for instance a human user, the agent itself, or another component of the system.

7.5 Conclusions and further work

Since agents are autonomous and pro-active entities, their behaviour cannot be completely predicted in open complex systems. This unpredictable behaviour can put at risk the integrity of a system. Thus, some mechanism is needed to guarantee the integrity of the system. Such a mechanism can be divided into two parts, the clear statement of what is considered to be *correct* behaviour, and a form of checking that any agent behaviour complies with that statement. In this chapter, we have presented our approach for the construction of these two parts of such a mechanism. For the first part, we proposed a model for specifying open multi-agent systems.

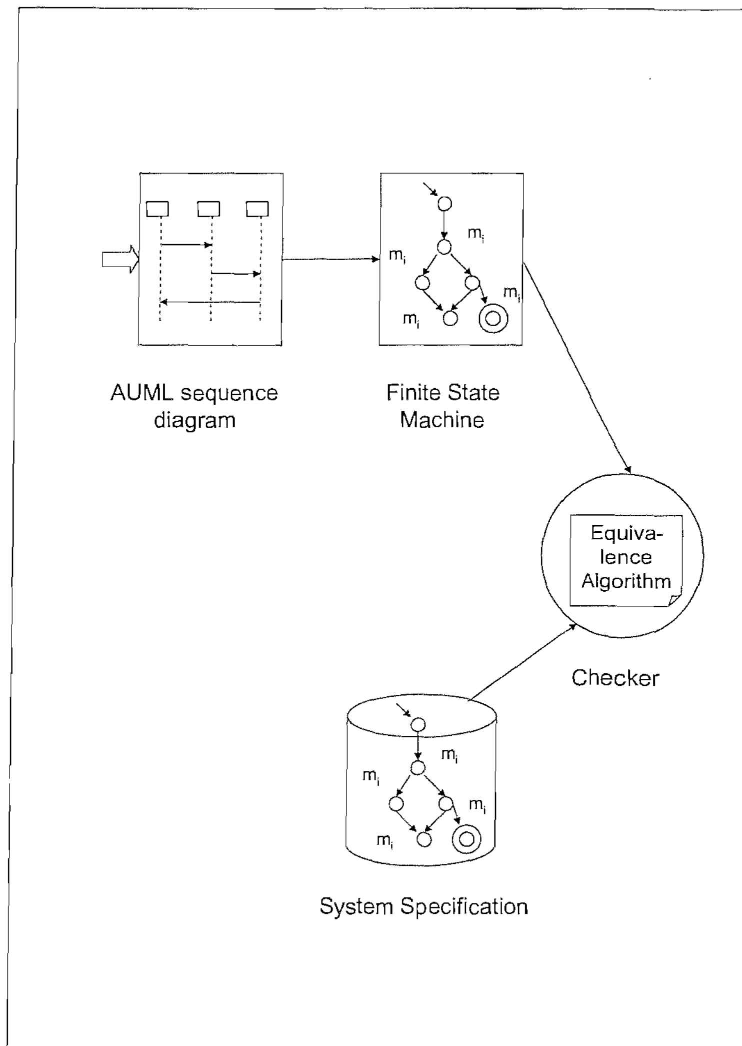

FIGURE 7.24: Checking protocol compliance

This specification of a system states what a valid agent behaviour is, regardless of its the actual implementation. Based on this model of specification, for the second part we presented a high-level design for checking the compliance of static aspects of the specification. The checking of the *dynamic* aspects of the specification, however, requires additional considerations, as sketched below.

Dynamic constraints must be continuously checked through the execution of a system. As stated earlier, we represent dynamic constraints by means of dynamic organisational rules. Since protocols and activities can be executed at any time, just as resources can be modified, any organisational rule involving at least one of these elements is considered to be dynamic, and must be continuously monitored to check its compliance. For the purpose of checking these dynamic characteristics, the monitor must be upgraded in several directions. First, its database must contain the organisational rules to be checked, and procedures to manipulate them. Such manipulation goes from basic tasks such as retrieval, or deciding if current conditions observe a

INPUTS:
a sequence of messages corresponding to a protocol

OUTPUT:
a finite state machine representing the sequence

ALGORITHM:

```
create new state
start_state = new state
current_state = new state
for all message in the sequence diagram
    if message is simple then
        create new_state
        create new transition from current_state to new_state
            with the communicative act of the message
        current_state = new_state
    else if the message connector == OR then
        create new_state
        create a new transition from current_state to new_state
            with null symbol
        create a new transition from new_state to itself
            for each concurrent message, using their
            communicative acts
        current_state = new_state
    else if the message connector == XOR then
        create new_state
        create a new transition from the current_state to new_state
            for each concurrent message, using their
            communicative acts
        current_state = new state
    else if the message connector == AND then
        create new_state
        for each possible permutation of the concurrent messages do
            create a path of transitions from current_state to new_state
                with the communicative acts, as they appear in the permutation
        end for
        current_state = new_state
    end if
end for
```

FIGURE 7.25: **Algorithm:** TRANSFORMING A SEQUENCE DIAGRAM INTO A FSM

rule (rule interpretation and evaluation), to more sophisticated tasks like indicating the reason for failure of a rule, or detecting contradictory or redundant rules.

Second, since dynamic organisational rules are constraints on the elements of system (roles, protocols, activities and resources), there must be a way to continually sense conditions related to these elements; for example, the order in which protocols are performed. The obvious way to do this is by storing in a single repository all the relevant information, namely the protocols and activities executed, together with their parameters. (The information dealing with the roles is stored in the monitor's database and the information about resources can be obtained from the parameters, assuming that each resource in the system can be uniquely identified.) This centralised approach, however, presents several disadvantages in terms of robustness and efficiency, since failures in this component may result in a total breakdown of the checking process. Also, concentrating such a large number of messages in one component may cause communication bottlenecks to appear. In order to simplify the design, the checking of dynamic organisational rules can be taken out of the monitor, and assigned to a new component, the *warden*, as follows.

The warden is in charge of ensuring that dynamic organisational rules are observed during the execution of the system. To accomplish this task, the warden is provided with relevant information from the monitor, the agents of the system, and from components called *collectors*, whose task is to filter the information provided by the agents. Figure 7.26 shows the overall operation of these components for checking dynamic organisational rules. First, when agents execute an activity or a protocol, they notify the collectors of the type of protocol or activity and the parameters used. When the collectors receive this information, they decide if it is relevant to the checking procedure, in which case they send it to the warden. The decision is made by employing a repository of organisational rules elements (*ORE* in the figure) which contains all the elements, and only those elements, of the system involved in one or more organisational rules. Note that to carry out this task, the collectors only need a list of organisational rules elements, so they can be destroyed and created as many times as needed. After receiving information from any collector, the warden updates its storage information (*IN* in the figure) and evaluates the corresponding organisational rules to verify that they have not been violated. The monitor provides the warden with the information regarding roles.

Some aspects of this design, as well as of the corresponding to static conditions, require further work to increase their applicability. The main aspects still to address are as follows.

First, since our design does not show, in detail, how to accomplish the monitoring of dynamic specifications, a more detailed design is needed. Second, assuming that the violations to the specification can be detected, some *policies* are needed to deal with them, as well as mechanisms to enact these policies. For example, when an attempt to violate a specification has arisen, the actions in question can just be ignored, or the involved agents can be notified about the violation and, additionally, about the conditions that originated it. In this case, the study of such policies and the design of such mechanisms are needed. Finally, it is necessary to determine the way these mechanisms affect the normal operation of a system, in terms of efficiency and robustness.

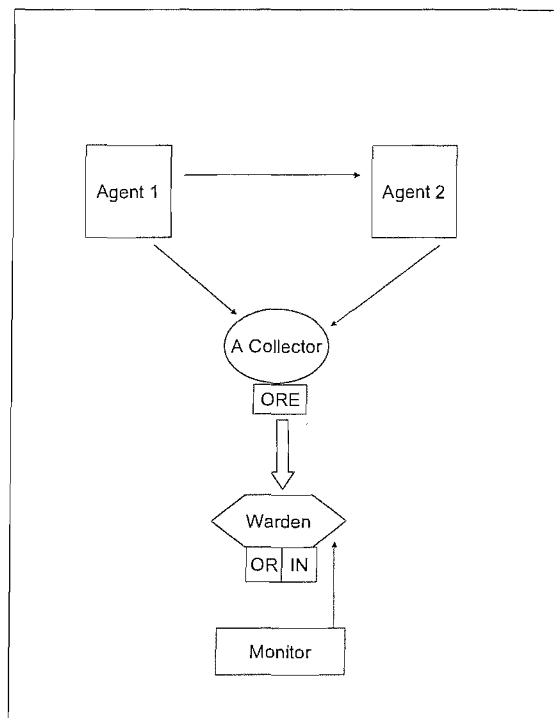

FIGURE 7.26: The operation of the warden

In summary, although guaranteeing the integrity of an open system is an essential part of the system life cycle, it has been largely unconsidered in agent research. As far as we are aware, this is the first attempt to solve this problem in a general form. Specifically, our contributions for the solution of this problem are the following.

- We have created a model for the specification of open multi-agent systems. This model is based on organisational concepts, can be instantiated for a specific system for describing the facilities provided by a system, and the way to access them, as well as for establishing the valid behaviour of the agents of the system. Such a description is important for the construction of new agents joining the system, and also forms a basis for monitoring that valid behaviours are observed during the operation of the system.
- We have created a classification of the conditions checked at run-time. This classification differentiates between static and dynamic conditions, which is essential for the design of run-time mechanisms that monitor the compliance of the conditions.
- We have designed a procedure for analysing equivalence between two protocols. Due to the use of AUML concurrency operators, the sequence of messages in a given protocol can be described in different ways. Thus, when monitoring the compliance of agents joining a system, a procedure is needed to analyse if the description of a protocol, as implemented by the agent, corresponds to the same protocol, as described in the specification. In this chapter, we have presented a procedure, based on the equivalence of non-deterministic finite automata that decides if two sequences of messages refer to the same protocol.

- We constructed a high level design for monitoring the compliance of static conditions. This design considers the information required for the monitoring, as well as procedures and components to accomplish it. The information required about the system and agent is based on the model of specification.

Chapter 8

Conclusions and future work

8.1 Summary

In this thesis, we have sought to address the problems involved in taking agent-oriented methodologies to a point where they can be used effectively in the development of open systems. With this aim, in this thesis we have sought to develop constructs and tools, as follows.

First, we have dealt with the development of a framework for constructing organisational patterns, which are representations of standard organisations, and are used for supporting the organisational design of multi-agent systems. Included in this development is the creation of a model for describing organisations.

Second, we have dealt with modelling the internal composition of agents, for which we have presented a means to develop agent architectural patterns to enable the incorporation of agent architectures into an agent design process. This includes techniques for obtaining the elements required to use the pattern from the results obtained by the design process. We have also developed a catalogue of such patterns, populated with different instantiations of agent architectures. By providing patterns for different architectures in this way, the catalogue avoids dependency on a specific architecture.

Next, we have described a methodological process that incorporates the tools mentioned above (organisational and architectural patterns), and that includes the use of iterations for decomposing the development of a system into more manageable units. This methodological process has been exemplified and assessed by means of a non-trivial case study, taken from an independent source.

Finally, we have presented a model for the specification of open systems which, when instantiated for a particular system, produces a specification whose compliance helps to ensure the correct operation of the system. This specification also describes the facilities offered by the system, and a means to access them, which is necessary for the construction of new agents joining the system.

8.2 Contributions

In addressing the problems involved in taking agent-oriented methodologies to a point where they can be used effectively in the construction of open systems, we have developed a series of ideas which have led to the construction of specific techniques and tools, each addressing a particular problem. These techniques and tools constitute the main contributions of this thesis and are described in the following.

1. LEVOR, a language for expressing organisational rules was created. Organisational rules are restrictions on how the elements of an organisation relate, and are used for specifying the behaviour of a system. However, in order to be effective, the description of organisational rules need to be *exact*. LEVOR is a language that allows the expression of organisational rules in such a way that their meaning is exact. Additionally, LEVOR is intuitive, easy to use, and can be extended to consider unforeseen situations. Being a computable language, LEVOR can be used to evaluate organisational rules at run-time, for monitoring their compliance during the operation of a system.

2. A model for characterisation and description of organisational structures was developed. Organisational structures are usually described by informal methods, such as using plain English or figures. Although useful, these methods are imprecise for specifying the architecture of a system — as in agent-oriented methodologies based on organisational concepts — or for creating representations of organisational structures — as in catalogues of organisational patterns. In this thesis, we have presented a model that allows the exact and complete representation of the components of an organisational structure, and from which a graphical representation can be obtained. In particular, the model provides a characterisation of the control regime of an organisational structure which identifies and classifies the most common types of control relationships.

3. A layout of organisational patterns was constructed. A pattern layout provides a means for describing the problem addressed by a pattern, the context in which this problem arises, and a solution to solve it. In particular, this layout of organisational patterns provides a way to describe patterns of organisational structures, and is formed of sections commonly found in any pattern layout, as well as sections specific for the description of organisational structures (which are based on the model for characterisation of organisational structures mentioned above). This layout can be used in the construction of catalogues of organisational patterns that support the organisational design of multi-agent systems, as was described in the methodological process developed in this thesis, and exemplified in associated the case study.

4. Instantiations of the layout for organisational patterns were developed. Three instantiations of the layout for organisational patterns were presented in this thesis, which served as examples of the use of the layout, and also as the basis for the construction of a catalogue of organisational patterns. Since these instantiations are pre-defined solutions that

can be re-utilised in different applications, they reduce development time by avoiding the need to begin the design from scratch. They also facilitate the design of a system, since the developer can focus only on those aspects specific for the application in question.

5. A mechanism for incorporating the use of agent architectures into the agent design phase was created. Agent design is an essential phase in the development of a multi-agent system, but this is typically not considered in several agent-oriented methodologies. In this thesis, we have not only considered this phase, but we have also provided tools to support it. These tools are based on the use of architectural patterns, and consist of a number of patterns that correspond to well known architectures, procedures for incorporating these patterns into the agent design phase of a methodological process, and methodological guidelines for the generation of new patterns. The architectural patterns provided can be used to significantly speed up the design of an agent, as was shown in the the case study, since the procedures help to match the inputs of the target pattern to the requirements provided by a methodological approach, and the guidelines assist in the process of creating architectural patterns for other agent architectures.

6. A methodological process for the design of multi-agent systems was developed. Although based on an existing methodology (Gaia), this methodological process completes and extends it with novel and valuable contributions. First, it incorporates an agent design phase, which includes models and activities to produce them. Second, integrated within the process is the use of organisational and architectural patterns, that speed up the development of a system by providing pre-defined solutions to the problems of determining the organisational structure of the system, and the internal composition of each agent of the system, respectively. Finally, the process incorporates an iterative approach that decomposes the development of a system into simpler, more manageable units, and allows the generation of executable versions of the system for each unit. A decomposition like this serves to obtain user feedback from early stages of the development and, in consequence, decreases the risk of building *the wrong* system.

7. A model for the specification of open multi-agent systems was created. This model provides a means to create specifications for open systems. Specifications are essential in open systems because they state the facilities provided by the system and the way to access them, which is important for the construction of new agents incorporating to the system. Additionally, specifications establish what is considered a valid behaviour of the agents of a system, and thus form the basis for monitoring that these are observed during the operation of the system.

8. A procedure for analysing equivalence between two protocols was designed. An important aspect of the specification of an open multi-agent systems refers to the protocols of agent interaction. To this end, the specification establishes, for each protocol, which sequences of messages are considered valid. However, depending on the way in which these sequences are described, it can be the case that one sequence has associated more than one

description. In this case, when monitoring the compliance of agents joining the system, a procedure is needed to analyse if the description of a protocol as implemented by the agent, corresponds to the same protocol, as described in the specification. In this thesis, we have presented a procedure, based on the equivalence of non-deterministic finite automata, that decides if two sequences of messages refer to the same protocol.

LEVOR can be used for the analysis and evaluation of organisational rules, for instance, to monitoring their compliance at run-time. LEVOR and the model for organisational structures, together, can be used to describe organisational structures. The descriptions obtained in this way can be used for different purposes, for example, for documenting a system or, as in this thesis, for the creation of organisational patterns.

The organisational patterns included in the thesis, and other possible organisational patterns generated by using the template, support the organisational design of a system and promotes re-utilisation. Similarly, the architectural patterns support the design of agents in a system, and help to avoid dependence on specific architectures.

The methodological process helps to manage the development cycle, with emphasis on controlling the risk of developing the wrong system and on keeping the project on time. Finally, the specification of open systems supports freedom to choose the most convenient form of development, and is also useful in monitoring that the behaviour of agents does not put at risk the integrity of a system.

8.3 Limitations

In spite of its contributions and benefits, there are some limitations in this work, as described below. First, the tools provided in this thesis are potentially suitable to be used as part of other methodologies, or even as stand-alone techniques, but additional work is needed to achieve this. In the case of organisational patterns, their description is tied to the characterisation of roles and protocols used in Gaia, which is not the same for other methodologies. However, the simplicity and the neutrality of these characterisations would facilitate the adaptation of the patterns. The case of the architectural patterns is slightly different, since they rely on agent architectures which are independent of any specific methodology. As for the methodological process, the basic idea of decomposing the development into iterations can be easily translated to other processes, especially for those methodologies that are extensions of Gaia (such as Roadmap [77]) or similar in structure to Gaia (such as SODA [99]).

Second, regarding the patterns, both organisational and architectural, although we have provided a layout, and some instantiations, the catalogues are immature, contain only a small number of patterns, and would benefit from refinement by the type of feedback obtained by recurrent use. In general, software patterns achieve maturity by constant use and by incorporating feedback provided by their users. In the long term, this process of utilisation and feedback incorporation

enhances a pattern with features that have proven to be useful in a significant number of applications, and eliminates from the pattern features that are particular to only a reduced number of situations. However, since the multi-agent approach is a relatively new approach, this maturity is not likely to occur in the short term. Nevertheless, this process can be accelerated by the construction of tools such as those provided in this thesis.

Third, the methodological process presented here does not consider in detail the phases of requirements analysis and implementation. These phases play an important role in development, the former by collecting and organising the information needed to understand the problem and the objectives pursued by the system, and the latter by providing the ultimate product of the development process, which is an executable version of the system. Thus, it is important to enhance the process with a complete consideration of these phases. The modular form in which the process is organised, however, lead us to believe that the incorporation of these phases can be achieved without significantly affecting the existing phases.

Fourth, regarding the monitoring of open multi-agent systems, we have provided a high-level design for checking the compliance of dynamic specifications. This high-level design, however, is far from a design from which a straightforward implementation could be obtained, since it does not provide solutions to problems such as how to obtain the information needed, how to minimise the traffic of information, what type of mechanism should be used to analyse if compliance is achieved, and what to do when a violation has been detected. We have, however, established the basis on which solutions to this problems can be devised, such as the model of specification, the differentiation between static and dynamic specifications, and a manipulable language for expressing this type of specification.

8.4 Future work

There are some areas of this thesis that would provide valuable benefits if they are subject of further work. In the following we describe these areas and briefly outline the work required for their completion.

The catalogues of organisational patterns and architectural patterns are potentially suitable for being used as stand-alone tools, or as part of other methodologies, for which some additional work is required. To begin with, those aspects of the patterns layout that depend on a particular methodological approach must be identified. For example, in the case of organisational patterns, these aspects include the form in which the elements of an organisation (such as roles and protocols) are described. These aspects can then be either replaced or adapted to fit other methodologies based on organisational concepts. Moreover, it can be possible to construct *families of layouts* that can be customised for a specific methodology.

In the case of architectural patterns, the process of adaptation to other methodologies can be facilitated by the absence of an agent design phase in many methodologies, and the fact that

agent architectures, on which the patterns are based, are *standard* tools, which do not depend on the methodology used. For those methodologies which do not include an agent design phase, the incorporation of the whole agent design phase presented in this thesis (including architectural patterns) can be a better option. For those methodologies which already include an agent design phase, only the procedures for generating the inputs to the patterns, from the outputs provided by the methodology in question, would require adaptation. Nevertheless, it must be noted that our architectural patterns use object-oriented notation and techniques, which can be unsuitable for some applications.

Further work is also needed for using organisational patterns in the development of large systems, in particular, when comparing different patterns for determining the pattern that best models a given system. In the process presented in this thesis, such comparison largely relies on *visual* examination, which can be adequate for small-sized systems and a small number of patterns. However, for systems involving a large number of roles and interactions, or when the number of patterns is significant, it is difficult to compare the structures visually, thus making a fully automated comparison process necessary. Such a process can be based on the model for organisational structures presented here.

For a complete coverage of the development cycle, the requirements analysis and implementation phases need to be incorporated into the methodological process presented in this thesis. The requirements analysis phase can be based on those requirements analysis techniques that have already been applied to agent-based methodologies, for example, goal-oriented requirements engineering [22], and agent-oriented requirements engineering [130]. On the other hand, the incorporation of the implementation phase requires the selection of some specific platforms, and of methodological guidelines to obtain a detailed specification, from design models, for these platforms.

Additionally, further work is required for the development of a graphical tool for supporting the different phases of the methodology. Such a tool is needed to facilitate and speed up the application of the methodological process (including the production of models), as well as for detecting inconsistencies and conflicts between the models, and generating documentation.

Finally, further work is required to produce a more detailed design for monitoring the compliance of dynamic social constraints. This design must include mechanisms for the analysis and evaluation of the constraints, for obtaining the information needed for such analysis, and must provide strategies to follow when a failure in the compliance with the constraints is detected.

8.5 Concluding remarks

In spite of its multiple benefits, agent-based computing has not received widespread uptake, particularly in sectors such as industry and commerce. This can be explained as a combination of several factors, such as the relative youth of the approach, and the natural resistance to change of

human organisations. However, the lack of maturity present in current agent-based software engineering also negatively affects its acceptance. Although overcoming this immaturity is partly a natural process that requires time, it also requires the development of tools and techniques to provide software developers with methodologies that are comprehensive and easy to apply.

Agent-based computing will play an important role in the computing world of the next years and, eventually, the multi-agent approach may become the dominant software paradigm for complex systems development. These are not only expectations, but predictions based on current evidence. However, the rate at which the multi-agent approach might become dominant largely depends on the rate at which agent-oriented software engineering reaches the required maturity. We believe that work like that presented in this thesis is important as a step even to approach such levels of maturity, especially if we are to realise true industrial take-up and application.

Bibliography

[1] Gregory D. Abowd and Elizabeth D. Mynatt. Charting past, present, and future research in ubiquitous computing. *ACM Transactions on Computer-Human Interaction*, 7(1):29–58, 2000.

[2] Magenta Multi agent Platform. http://www.magenta-technology.com/technology/multiagent/, 2006.

[3] Agentis. http://www.agentissoftware.com/en/solutions/components.jsp, 2006.

[4] ANSI. http://logic.stanford.edu/kif/specification.html, 1995.

[5] Ofer Arazy and Carson Woo. Analysis and design of agent-oriented information systems. *The Knowledge Engineering Review*, 17(3):215–260, 2002.

[6] Y. Aridor and D. Lange. Agent design patterns: Elements of agent application design. In *Autonomous Agents (Agents'98)*. ACM Press, 1998.

[7] G. Beavers and H. Hexmoor. Teams of agents. In *IEEE International Conference on Systems, Man, and Cybernetics*, volume 1, 2001.

[8] Ken Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change, Second Edition*. Addison-Wesley, 2004.

[9] Paolo Bresciani, Anna Perini, Paolo Giorgini, Fausto Giunchiglia, and John Mylopoulos. Tropos: An agent-oriented software development methodology. *Autonomous Agents and Multi-Agent Systems*, (8):203–236, 2004.

[10] Rodney A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2(1):14–23, 1986.

[11] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture*. Wiley, 1996.

[12] Stefan Bussmann, Nicholas Jennings, and Michael Wooldridge. *Multiagent Systems for Manufacturing Control*. Springer, 2004.

[13] Giovanni Caire, Wim Coulier, Francisco Garijo, Jorge Gomez-Sanz, Juan Pavon, Paul Kearney, and Philippe Massonet. *Methodologies and Software Engineering for Agent Systems*, chapter The MESSAGE methodology. Kluwer Academic, 2004.

[14] K. M. Carley and L. Gasser. *Multiagent Systems*, chapter Computational Organization Theory. MIT Press, 1999.

[15] Jaelson Castro, Manuel Kolp, and John Mylopoulos. Towards requirements-driven information systems engineering: The tropos project. *Information Systems*, 27(6):365–389, 2002.

[16] Luca Cernuzzi and Franco Zambonelli. Dealing with adaptive multiagent systems organizations in the gaia methodology. In *6th International Workshop on Agent-Oriented Software Engineering (AOSE 2005)*, 2006.

[17] A. Chavez and P. Maes. Kasbah: an agent marketplace for buying and selling goods. In *First International Conference on the Practical Application of Intelligent Agents and Multi-agent Technology PAAM'96*, pages 75–90. Practical Application Company, 1996.

[18] L. Chung, B. Nixon, E. Yu, and J. Mylopoulos. *Non-functional Requirements in Software Engineering*. Kluwer Academin Press, 2000.

[19] Alistair Cockburn. *Agile Software Development*. Addison-Wesley, 2001.

[20] Massimo Cossentino, Piermarco Burrafato, Saverio Lombardo, and Luca Sabatucci. Introducing pattern reuse in the design of multi-agent systems, 2002.

[21] R. Scott Cost, Ye Chen, Tim Finin, Yannis Labrou, and Yun Peng. *Using Colored Petri Nets for Conversation Modeling*, volume 1916 of *Lecture Notes in AI*, pages 178–192. Springer-Verlag, 2000.

[22] A. Dardenne, A. van Lamsweerde, and S. Fickas. Goal-directed requirements acquisition. *Science of Computer Programming*, (20):3–50, 1993.

[23] W. Davis and P. Edwards. AGENT-K: An integration of AOP and KQML. In *Proceedings of the CIKM'94 Workshop on Intelligent Agents*, 1994.

[24] Scott A. DeLoach. Multiagent systems engineering: A methodology and language for designing agent systems. In *Agent-Oriented Information Systems (AOIS'99)*, pages 45–57, 1999.

[25] Scott A. DeLoach. Modeling organizational rules in the multiagent systems engineering methodology. In R. Cohen and B. Spencer, editors, *15th Canadian Conference on Artificial Intelligence*, volume 2338 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 2002.

[26] D. Deugo, M. Weiss, and E. Kendall. *Coordination of Internet Agents: Models, Technologies and Applications*, chapter Reusable Patterns for Agent Coordination. Springer, 2001.

[27] Virginia Dignum, Javier Vazquez-Salceda, and Frank Dignum. Omni: Introducing social structure, norms and ontologies into agent organizations. In *Programming Multi-Agent Systems: Second International Workshop ProMAS 2004*, volume 3346 of *Lecture Notes in Artificial Intelligence*. Springer, 2005.

[28] Mark d'Inverno, David Kinny, Michael Luck, and Michael Wooldridge. A formal specification of dMARS. In M.P. Singh, A. Rao, and M.J. Wooldridge, editors, *Intelligent Agents IV. Proceedings of the Forth International Workshop on Agent Theories, Architectures, and Languages*, volume 1365 of *Lecture Notes in Artificial Intelligence*, pages 155–176. Springer-Verlag, 1998.

[29] Mark D'Inverno, Michael Luck, Michael Georgeff, David Kinny, and Michael Wooldridge. The dMARS architecture: A specification of the distributed milti-agent reasoning system. *Autonomous Agents and Milti-Agent Systems*, (9):5–53, 2004.

[30] Eric Dubois, Philippe Du Bois, and Michal Petit. Agent-oriented requirements engineering: A case study using the ALBERT language. In *Preceedings of the Fourth International Working Conference on Dynamic Modelling and Information Systems DYNMOD'94*, pages 205–238, 1994.

[31] J. Bradshaw (ed.). *Software Agents*. MIT Press, 1997.

[32] Innes Ferguson. *TouringMachines: An Architecture for Dynamic, Rational, Mobile Agents*. PhD thesis, University of Cambridge, 1992.

[33] Innes Ferguson. Integrated control and coordinated behaviour: a case for agent models. In *Intelligent Agents ECAI-94*, pages 203–218. Springer-Verlag, 1995.

[34] Tim Finin, Yannis Labrou, and James Mayfield. *Software Agents*, chapter KQML as an agent communication language. MIT Press, 1995.

[35] FIPA. http://www.fipa.org/, 1999.

[36] FIPA-OS. http://fipa-os.sourceforge.net/, 2000.

[37] K. Fischer. Agent-based design of holonic manufacturing systems. *Journal of Robotics and Autonomous Systems*, 1999.

[38] Klaus Fischer, Michael Schillo, and Jörg H. Siekmann. Holonic multiagent systems: A foundation for the organisation of multiagent systems. In *HoloMAS*, pages 71–80, 2003.

[39] Michael Fisher. Representing and executing agent-based systems. In M. Wooldridge and N. Jennings, editors, *Intelligent Agents*, volume 890, pages 307–323. Springer-Verlag, 1995.

[40] Michael Fisher, Jorg Muller, Michael Schroeder, Gerd Wagner, and Geof Staniford. Methodological foundations for agent-based systems. In *Proceedings of the UK Special Interest Group on Foundations of Multi-Agent Systems (FOMAS)*, volume 12. The Knowledge Engineering Review, 1997.

[41] Stanford Center for Design. http://java.stanford.edu/java_agent/html, 2000.

[42] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *International Journal of Supercomputer Applications*, 15(3), 2001.

[43] M. Fowler and K. Scott. *UML Destilled: Applying the Standard Object Modeling Language*. Addison-Wesley, 1997.

[44] M. Fox, M. Barbuceanu, M. Gruninger, and J. Lin. *Simulating Organizations*, chapter An Organizational Ontology for Enterprise Modeling. AAAI Press/The MIT Press, 1998.

[45] Mark S. Fox. An organizational view of distributed systems. *IEEE Transactions on Systems, Man, and Cybernetics*, 11(1):70–80, 1981.

[46] Lost Wax Agent Framework. http://www.lostwax.com/agents/framework/, 2006.

[47] Stan Franklin and Art Graesser. Is it an agent or just a program? a taxonomy for autonomous agents. In *Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages (ATAL 96)*, volume 1193, pages 21–35. Springer-Verlag, 1996.

[48] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[49] A. Garcia, C. Chavez, and R. Choren. An aspect-oriented modeling framework for MAS design. In *7th Workshop on Agent-Oriented Software Engineering, AAMAS'06*, 2006.

[50] Juan Garcia-Ojeda, Alvaro Arenas, and Jose Perez-Alcazar. Paving the way for implementing multiagent systems. In *6th International Workshop on Agent-Oriented Software Engineering (AOSE 2005)*, 2006.

[51] Michael P. Georgeff and Amy L. Lansky. Reactive reasoning and planning. In *AAAI*, pages 677–682, 1987.

[52] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning*. The Morgan Kaufmann Series in Artificial Intelligence. Morgan Kaufmann, 2004.

[53] N. Glaser. *Contribution to Knowledge Modelling in a Multi-agent Framework*. PhD thesis, Universit Henry Poincar, 1996.

[54] Jorge Gomez-Sanz and Juan Pavon. Meta-modelling in agent oriented software engineering. In F. Garijo, J. Riquelme, and M. Toro, editors, *Advances in Artificial Intelligence (IBERAMIA 2002)*, pages 606–615. Springer-Verlag, 2002.

[55] Jorge Gomez-Sanz and Juan Pavon. Agent oriented software engineering with INGENIAS. In V. Marik, J. Muller, and M. Pechoucek, editors, *Multi-Agent Systems and Applications III*, pages 394–403. Springer-Verlag, 2003.

[56] Jorge Gonzalez-Palacios and Michael Luck. A framework for patterns in gaia: A case study with organisations. In *Agent-Oriented Software Engineering V (AOSE 2004)*, volume 3382 of *Lecture Notes in Computing Science*, pages 174–188. Springer, 2005.

[57] Jorge Gonzalez-Palacios and Michael Luck. Towards compliance of agents in open multi-agent systems. In *Software Engineering for Large Scale Multi-agent Systems 2006 SELMAS V*, Lecture Notes in Computing Science. Springer, 2007. To appear.

[58] D. Grossi, F. Dignum, V. Dignum, M. Dastani, and L. Royakkers. Structural aspects of the evaluation of agent organizations. In *Proceedings of the Workshop on Coordination, Organization, Institutions and Norms in Agent Systems (COIN @ ECAI 06)*, 2006.

[59] Object Management Group. http://www.omg.org, 2006.

[60] W3C group. http://www.w3.org/tr/wsci/, 2002.

[61] Afsaneh Haddadi and Kurt Sundermeyer. *Foundations of Distributed Artificial Intelligence*, chapter Belief-Desire-Intention Agent Architectures. John Wiley and Sons, 1996.

[62] S. Hayden, C. Carrick, and Q. Yang. Architectural design patterns for multiagent coordination. In *International Conference on Agent Systems '99 (Agents'99)*, 1999.

[63] G. Hohpe and B. Woolf. *Enterprise Integration Patterns*. Addison-Wesley, 2003.

[64] Bryan Horling and Victor Lesser. A survey of multi-agent organizational paradigms. *The Knowledge Engineering Review*, 0(0):1–24, 2005.

[65] Zimmermann Hubert. The ISO model of architecture for open systems interconnection. *IEEE Transactions on Communications*, 28(4):425–432, 1980.

[66] Marc P. Huget, James Odell, and Bernhard Bauer. *Methodologies and Software Engineering for Agent Systems*, chapter The AUML approach. Kluwer Academic, 2004.

[67] Carlos A. Iglesias, Mercedes Garijo, and Jose C. Gonzalez. A survey of agent-oriented methodologies. In J.P. Muller, M.P. Singh, and A. Rao, editors, *Intelligent Agents V. Proceedings of the Fifth International Workshop on Agent Theories, Architectures, and Languages (ATAL-98)*, volume 1555, pages 317–330. Springer-Verlag, 1999.

[68] Carlos A. Iglesias, Mercedes Garijo, Jose C. Gonzalez, and Juan R. Velasco. Analysis and design of multiagent systems using MAS-CommonKADS. In M.P. Singh, A. Rao, and M.J. Wooldridge, editors, *Intelligent Agents IV. Proceedings of the Fourth International Workshop on Agent Theories, Architectures, and Languages (ATAL-97)*, volume 1365, pages 313–326. Springer-Verlag, 1998.

[69] Ivar Jacobson, J. Rumbaugh, and G. Booch. *The Unified Software Development Process*. Addison-Wesley, 1999.

[70] JADE. http://sharon.cselt.it/projects/jade/, 1999.

[71] N. R. Jennings, S. Parsons, C. Sierra, and P. Faratin. Automated negotiation. In *Proceedings of the 5th International Conference on Practical Application of Intelligent Agents and Multi-Agent Systems (PAAM-2000)*, pages 23–30, 2000.

[72] Nicholas R. Jennings. Specification and implementation of a belief, desire joint-intention architecture for collaborative problem solving. *Journal of Intelligent and Cooperative Information Systems*, 2(3):289–318, 1993.

[73] Nicholas R. Jennings. Controlling cooperative problem solving in industrial multi-agent systems. *Artificial Intelligence*, 2(75):195–240, 1995.

[74] Nicholas R. Jennings. On agent-based software engineering. *Artificial Intelligence*, 117(2):277–296, 2000.

[75] Nicholas R. Jennings. An agent-based approach for building complex software systems. *Communications of the ACM*, 44(4):35–41, 2001.

[76] Mark W. Johnson, Peter McBurney, and Simon Parsons. A mathematical model of dialog. *Electr. Notes Theor. Comput. Sci.*, 141(5):33–48, 2005.

[77] Thomas Juan, Adrian Pearce, and Leon Sterling. Roadmap: Extending the gaia methodology for complex open systems. In *AAMAS '02*. ACM, 2002.

[78] Paul Kearney, Jamie Stark, Giovanni Caire, Francisco J. Garijo, Jorge J. Gomez Sanz, Juan Pavon, Francisco Leal, Paulo Chainho, and Philippe Massonet. Message: Methodology for engineering systems of software agents. Technical Report EDIN 0223-0907, Eurescom, 2001.

[79] E. Kendall. Role models: Patterns of agent system analysis and design. *BT Technology Journal*, 17(4):46–57, 1999.

[80] D. Kinny, M. Georgeff, and A. Rao. A methodology and modelling technique for systems of BDI agents. In W. van der Velde and J. Perram, editors, *Agents Breaking Away: Proceedings of the Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW-96)*, volume 1038 of *Lecture Notes in Artificial Intelligence*, pages 56–71. Springer-Verlag, 1996.

[81] M. Kolp, J. Castro, and J. Mylopoulos. A social organization perspective on software architectures. In *First Int. Workshop From Software Requirements to Architectures*, 2001.

[82] Manuel Kolp, Paolo Giorgini, and John Mylopoulos. A goal-based organizational perspective on multi-agent architectures. In J. J. Ch. Meyer and M. Tambe, editors, *Intelligent Agents VIII, LNAI 2333*, pages 128–140. Springer-Verlag, 2002.

[83] Harry Lewis and Christos Papadimitriou. *Elements of the Theory of Computation*. Prentice-Hall, 1998.

[84] J. Lind. Patterns in agent-oriented software engineering. In Fausto Giunchiglia, James Odell, and Gerhard Weiss, editors, *Agent-Oriented Software Engineering III*, volume 2585 of *Lecture Notes in Computer Science*. Springer, 2003.

[85] Scott A. Loach, Mark F. Wood, and Clint H. Sparkman. Multiagent systems engineering. *International Journal of Software Engineering and Knowledge Engineering*, 11(3):231–258, 2001.

[86] Alessio Lomuscio, Marek Sergot, John-Jules Meyer, and Milind Tambe. On multi-agent systems specification via deontic logic. In *Intelligent agents VIII : agent theories, architectures, and languages (ATAL 2001)*, volume 2333 of *Lecture Notes in Computer Science*, pages 86–99. Springer, 2002.

[87] Fabiola Lopez, Michael Luck, and Mark D'Inverno. A normative framework for agent-based systems. In *Proceedings of the Symposium on Normative Multiagent Systems (NorMAS 05)*, pages 24–35, 2005.

[88] M. Luck, P. McBurney, O. Shehory, and S.Willmott. *Agent Technology: Computing as Interaction. A Roadmap for Agent Based Computing*. AgentLink III, 2005.

[89] Michael Luck. From definition to deployment: What next for agent-based systems? *The Knowledge Engineering Review*, 14(2):119–124, 1999.

[90] Michael Luck, Nathan Griffiths, and Mark d'Inverno. From agent theory to agent construction: A case study. In J.P. Muller, M.J. Wooldridge, and N.R. Jennings, editors, *Intelligent Agents III. Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages*, volume 1193 of *Lecture Notes in Artificial Intelligence*, pages 49–63. Springer-Verlag, 1997.

[91] Michael Luck, Peter McBurney, and Jorge Gonzalez-Palacios. Agent-based computing and programming of agent systems. In *Programming Multi-Agent Systems*, volume 3862 of *Lecture Notes in Artificial Intelligence*, pages 23–37. Springer, 2006.

[92] Michael M. Luck, Donald Ashri, and Mark D'Inverno. *Agent Based Software Development*. Artech House Publishers, 2004.

[93] Tambe M. Towards flexible teamwork. *Journal of Artificial Intelligence Research*, (7):83–124, 1997.

[94] Philippe Massonet, Yves Deville, and Cedric Neve. From aose methodology to agent implementation. In *AAMAS'02*, pages 15–19. ACM, 2002.

[95] P. Mathieu, J. Routier, and Y Secq. Dynamic organization in multi-agent systems. In *Proceedings of the First International Conference on Autonomous Agents and Multi-Agent Systems*, pages 451–452. ACM Press, 2002.

[96] Juan Pavon Mestras, Jorge Gomez Sanz, and Ruben Fuentes. http://grasia.fdi.ucm.es/ingenias/, 1999.

[97] Hyacinth S. Nwana and Divine T. Ndumu. A perspective on software agents research. *The Knowledge Engineering Review*, 14(2):125–142, 1999.

[98] James Odell, H. Van Dyke Parunak, and Bernhard Bauer. Representing agent interaction protocols in UML. In Paolo Ciancarini and Michael Wooldridge, editors, *Agent Oriented Software Engineering*, pages 121–140. Springer, 2001.

[99] Andrea Omicini. SODA: Societies and infraestructures in the analysis and design of agent-based systems. In P. Ciancarini and M. J. Wooldridge, editors, *Proceedings of the First International Workshop in Agent-oriented Software Engineering (AOSE–2000)*, volume 1957 of *Lecture Notes in Artificial Intelligence*, pages 185–194. Springer-Verlag, 2001.

[100] Gauthier Picard and Marie-Pierre Gleizes. *Methodologies and Software Engineering for Agent Systems*, chapter The ADELFE methodology. Kluwer Academic, 2004.

[101] LISA Project. http://www.lisa.sourceforge.net/, 2006.

[102] Tropos project. http://www.troposproject.org/, 2006.

[103] A. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In W. van der Velde and J. Perram, editors, *Agents Breaking Away: Proceedings of the Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW-96)*, volume 1038 of *Lecture Notes in Artificial Intelligence*, pages 42–55. Springer-Verlag, 1996.

[104] Anand S. Rao and Michael P. Georgeff. BDI agents: From theory to practice. In *ICMAS-95. Proceedings of the First International Conference on Multi-Agent Systems*, pages 312–319, 1995.

[105] P. Ricordel and Y. Demazeau. From analysis to deployment: A multi-agent plataform survey. In *Working notes of the First International Workshop on Engineering Societies in the Agents' World (ESAW-00)*, 2000.

[106] Juan Rodriguez-Aguilar, Francisco Martin, Pablo Noriega, Pere Garcia, and Carles Sierra. Towards a test-bed for trading agents in electronic auction markets. *AI Communications*, 11(1):5–19, 1998.

[107] Ekkart Rudolph, Jens Grabowski, and Peter Graubmann. Tutorial on message sequence charts (MSC). In *Proceedings of FORTE/PSTV'96 Conference*, 1996.

[108] Michael Schillo, Klaus Fischer, and Jörg H. Siekmann. The link between autonomy and organisation in multiagent systems. In *HoloMAS*, pages 81–90, 2003.

[109] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice Hall, 1996.

[110] Onn Shehory and Arnon Sturm. Evaluating agent-based systems modeling techniques. Technical Report TR-ISE/IE-003-2000, Faculty of Industrial Engineering and Management Technion - Israel Institute of Technology, 2000.

[111] Onn Shehory and Arnon Sturm. Evaluation of modelling techniques for agent-based systems. In J.P. Muller, Elisabeth Andre, Sandip Sen, and Claude Frasson, editors, *Agents 01. Proceedings of the Fifth International Conference on Autonomous Agents*, pages 624–631. ACM Press, 2001.

[112] Y. Shoham. Agent-oriented programming. *Artificial Intelligence*, 1(60):51–92, 1993.

[113] SICS. Traiding agent competition, 2006.

[114] A. Silva and J. Delgado. The agent pattern for mobile agent systems. In *3rd European Conference on Pattern Languages of Programming and Computing, EuroPLoP'98*, 1998.

[115] R. G. Smith. The contract net: A formalism for the control of distributed problem solving. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence (IJCAI-77)*, 1977.

[116] Agent Oriented Software. http://www.agent-software.com.au/shared/home/index.html, 1998.

[117] Rational Software. Rational unified process. Technical Report TP026B 11/01, Rational Software, 1998.

[118] Living Systems Technology Suite. http://www.whitestein.com/pages/solutions/ls ts.html, 2006.

[119] Paul Taylor, Peter Evans-Greenwood, and James Odell. Agents in the enterprise. In *Australian Software Engineering Conference ASWEC 2005*, 2005.

[120] British Telecom. http://www.labs.bt.com/projects/agents/zeus, 1999.

[121] L. van der Torre, J. Hulstijn, M. Dastani, and J. Broersen. Specifying multiagent organizations. In *Proceedings of the Seventh Workshop on Deontic Logic in Computer Science (Deon'2004)*, volume 3065 of *Lecture Notes in Computer Science*, pages 243–257. Springer, 2004.

[122] Wamberto Vasconcelos, Mairi McCallum, and Tim Norman. Modelling organisational change using agents. Technical Report AUCS/TR0605, Department of Computing Science, University of Aberdeen, 2006.

[123] Luis Erasmo Montealegre Vzquez and Fabiola Lpez y Lpez. An agent-based model for hierachical organizations. In *Proceedings of the Workshop on Coordination, Organization, Institutions and Norms in Agent Systems (COIN @ ECAI 06)*, 2006.

[124] Gerhard Weiss. Agent orientation in software engineering. *The Knowledge Engineering Review*, 16(4):349–373, 2001.

[125] M. Wellman. *Practical Handbook of Internet Computing*, chapter Online marketplaces. Chapman Hall & CRC press, 2004.

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

[127] Michael Wooldridge. Diversity and agent technology. *The Knowledge Engineering Review*, 14(2):151–152, 1999.

[128] Michael Wooldridge. *Multiagent Systems: a Modern Approach to Distributed Artificial Intelligence*, chapter Intelligent Agents. MIT Press, 1999.

[129] Michael Wooldridge and Nicholas R. Jennings. Intelligent agents: theory and practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995.

[130] E. Yu. Towards modelling and reasoning support for early-phase requirements engineering. In *Preceedings of 3rd IEEE International Symposium on Requirements Engineering RE'97*, pages 226–235, 1997.

[131] Franco Zambonelli, Nicholas R. Jennings, Andrea Omicini, and Michael Wooldridge. *Coordination of Internet Agents: Models, Technologies and Applications*, chapter Agent-Oriented Software Engineering for Internet Applications. Springer, 2001.

[132] Franco Zambonelli, Nicholas R. Jennings, and Michael Wooldridge. Organisational abstractions for the analysis and design of multi-agent systems. In *First International Workshop on Agent-Oriented Software Engineering*, pages 127–141, 2000.

[133] Franco Zambonelli, Nicholas R. Jennings, and Michael Wooldridge. Organisational rules as an abstraction for the analysis and design of multi-agent systems. *International Journal of Software Engineering and Knowledge Engineering*, 11(3):303–328, 2001.

[134] Franco Zambonelli, Nicholas R. Jennings, and Michael Wooldridge. Developing multiagent systems: The Gaia methodology. *ACM Transactions on Software Engineering and Methodology*, 12(3):317–370, 2003.

[135] H. Zhang and Victor Lesser. A dynamically formed hierarchical agent organization for a distributed content sharing system. In *Proceedings of the International Conference on Intelligent Agent Technology*, 2004.

[136] X. Zhang and D. Norrie. Holonic control at the production and controller levels. In *Proceedings of IMS 99*, pages 215–224, 1999.

Appendix A

The Conference Management System problem statement

The purpose of the system is to support the management of a medium to large conference. The management of a conference can be divided into several phases: submission, review, and notification. In the submission phase an open call is made for the authors to send their papers within a deadline; authors send their papers and receive a number which serves as confirmation of the reception.

During the review phase, the papers that will be presented in the conference are selected. The selection is supervised and coordinated by the Program Committee and the review of the papers is performed by referees. A number of papers are sent to each referee to review and each paper is reviewed by three referees. Since the committee members and the referees can be authors, some rules have to be observed, for example, a referee cannot review his own paper. The reviews are the base to decide if a paper is accepted or rejected.

Finally, in the notification phase each author is sent a notification containing the reviews, the decision of acceptance or rejection of his paper and, in the former case, a deadline to produce the final version of the paper. Next, the publisher has to collect all the final versions and print the proceedings.