MONOGRAFIES DE L'INSTITUT D'INVESTIGACIÓ
EN INTEL·LIGÈNCIA ARTIFICIAL
Number 19

Institut d'Investigació
en Intel·ligència Artificial

Consell Superior
d'Investigacions Científiques

Monografies de l'Institut d'Investigació en Intel·ligència Artificial

Num. 1 J. Puyol, *MILORD II: A Language for Knowledge–Based Systems*

Num. 2 J. Levy, *The Calculus of Refinements, a Formal Specification Model Based on Inclusions*

Num. 3 Ll. Vila, *On Temporal Representation and Reasoning in Knowledge–Based Systems*

Num. 4 M. Domingo, *An Expert System Architecture for Identification in Biology*

Num. 5 E. Armengol, *A Framework for Integrating Learning and Problem Solving*

Num. 6 J. Ll. Arcos, *The Noos Representation Language*

Num. 7 J. Larrosa, *Algorithms and Heuristics for Total and Partial Constraint Satisfaction*

Num. 8 P. Noriega, *Agent Mediated Auctions: The Fishmarket Metaphor*

Num. 9 F. Manyà, *Proof Procedures for Multiple-Valued Propositional Logics*

Num. 10 W. M. Schorlemmer, *On Specifying and Reasoning with Special Relations*

Num. 11 M. López-Sánchez, *Approaches to Map Generation by means of Collaborative Autonomous Robots*

Num. 12 D. Robertson, *Pragmatics in the Synthesis of Logic Programs*

Num. 13 P. Faratin, *Automated Service Negotiation between Autonomous Computational Agents*

Num. 14 J. A. Rodríguez, *On the Design and Construction of Agent-mediated Electronic Institutions*

Num. 15 T. Alsinet, *Logic Programming with Fuzzy Unification and Imprecise Constants: Possibilistic Semantics and Automated Deduction*

Num. 16 A. Zapico, *On Axiomatic Foundations for Qualitative Decision Theory - A Possibilistic Approach*

Num. 17 A. Valls, *ClusDM: A multiple criteria decision method for heterogeneous data sets*

Num. 18 D. Busquets, *A Multiagent Approach to Qualitative Navigation in Robotics*

Num. 19 M. Esteva, *Electronic Institutions: from specification to development*

Num. 20 J. Sabater, *Trust and reputation for agent societies*

Electronic Institutions: from specification to development

Marc Esteva

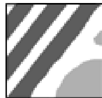
Foreword by Carles Sierra

2003 Consell Superior d'Investigacions Científiques
Institut d'Investigació en Intel·ligència Artificial
Bellaterra, Catalonia, Spain.

Series Editor
Institut d'Investigació en Intel·ligència Artificial
Consell Superior d'Investigacions Científiques

Foreword by
Carles Sierra
Institut d'Investigació en Intel·ligència Artificial
Consell Superior d'Investigacions Científiques

Volume Author
Marc Esteva
Institut d'Investigació en Intel·ligència Artificial
Consell Superior d'Investigacions Científiques



Institut d'Investigació
en Intel·ligència Artificial

Consell Superior
d'Investigacions Científiques

© 2003 by Marc Esteva
NIPO: 403-03-084-2
ISBN: 84-00-08156-0
Dip. Legal: B.43078-2003

All rights reserved. No part of this book may be reproduced in any form or by any electronic or mechanical means (including photocopying, recording, or information storage and retrieval) without permission in writing from the publisher.
Ordering Information: Text orders should be addressed to the Library of the IIIA, Institut d'Investigació en Intel·ligència Artificial, Campus de la Universitat Autònoma de Barcelona, 08193 Bellaterra, Barcelona, Spain.

Contents

Foreword **xi**

Abstract **xiii**

Acknowledgements **xv**

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Contributions . . . 3
1.3 Strucutre . . . 7

2 State of the Art **9**
2.1 Multi-agent Systems . . . 9
2.2 Agent Communication . . . 15
2.3 Agent Platforms . . . 19

3 Electronic Institutions **23**
3.1 Dialogic Framework . . . 24
3.2 Scene . . . 26
3.2.1 Variables . . . 29
3.2.2 Constraints . . . 30
3.3 Performative Structure . . . 33
3.4 Norms . . . 37
3.5 Electronic Institution . . . 39
3.6 Conclusions . . . 39

4 Formalising Institutions in Process Algebras **41**
4.1 Distributed winner determination process . . . 42
4.1.1 Organization of the auction room . . . 43
4.1.2 The π-calculus in brief . . . 45
4.1.3 Leader Election . . . 47
4.1.4 The Governor . . . 50
4.1.5 First-Price/Sealed-Bid . . . 53
4.1.6 Vickrey's auction . . . 55
4.1.7 Dutch auction . . . 55

4.1.8 English auction . . . 57
4.2 Conclusions . . . 60

5 Islander **61**
5.1 ISLANDER language definition . . . 61
5.1.1 Electronic Institution . . . 61
5.1.2 Dialogic Framework . . . 62
5.1.3 Ontology . . . 63
5.1.4 Performative Structure . . . 64
5.1.5 Scene . . . 66
5.1.6 Norms . . . 69
5.2 ISLANDER editor . . . 70
5.2.1 Islander editor modules . . . 70
5.2.2 Graphic User Interface . . . 73
5.3 Verification . . . 76
5.3.1 Integrity . . . 76
5.3.2 Liveness . . . 76
5.3.3 Protocol correctness . . . 84
5.3.4 Norm correctness. . . . 89
5.3.5 Model Checking Scenes . . . 89
5.4 Conclusions . . . 90

6 Social Layer for Electronic Institutions **93**
6.1 Institution architecture . . . 94
6.2 JADE in brief . . . 97
6.3 Social layer . . . 100
6.3.1 Governor . . . 101
6.3.2 Scene Management . . . 107
6.3.3 Transition management . . . 113
6.3.4 Norm management . . . 117
6.4 Development of agents for electronic institutions . . . 119
6.4.1 Agoric Market . . . 121
6.4.2 Logical formalism for Electronic institutions . . . 121
6.4.3 Synthesis of Agents . . . 122
6.4.4 Customising Synthesised Agents . . . 124
6.5 Conclusions . . . 127

7 Applications **129**
7.1 Auction house federation . . . 129
7.1.1 Dialogic Framework . . . 130
7.1.2 Performative Structure . . . 131
7.1.3 Auction Scene . . . 137
7.2 Conference Centre . . . 138
7.2.1 Dialogic framework . . . 139
7.2.2 Performative Structure . . . 140
7.2.3 Appointment Proposal Scene . . . 143

7.2.4 Norms 145
7.3 Conclusions 145

8 Conclusions **147**
8.1 Future Work 150

A ISLANDER specifications **155**
A.1 Auction house federation 155
A.2 Conferen Centre 165

List of Figures

3.1 Specification of the sealed-bid auction protocol 28
3.2 Graphical Elements of a Performative Structure 36

4.1 Organization of the agents in the auction room 44
4.2 Specification of the generic bidding resolution protocol 50
4.3 Example of collision resolution 51
4.4 The four steps of a bidding round 52
4.5 Specification of the processes FP-WaitingBid, FP-bid and FP-WaitingMessage. 54
4.6 Specification of the process FP-ResolveBids 56
4.7 Specification of the process DA-bid 57
4.8 Specification of the process EA-WaitingMessage 58
4.9 Specification of the process EA-ResolveBid 59

5.1 ISLANDER editor modules 70
5.2 ISLANDER Data Flow Diagram 71
5.3 ISLANDER Graphic User Interface 74
5.4 Fish Market Performative Structure 82
5.5 Depth-first search of the Fish Market performative structure . . 83
5.6 Auction scene following a Dutch protocol 87
5.7 Depth-first search for the acution scene 88

6.1 Institution architecture 95
6.2 Architecture of the JADE platform 98
6.3 Governor architecture. 102
6.4 Algorithm for saying an agent message 109
6.5 Examples of transitions. 115
6.6 Performative structure of the agoric market. 120
6.7 Simple Agora Room Scene 121
6.8 Representation of Agora Room Scene 122
6.9 Representation of Agoric Market electronic institution 122
6.10 Synthesised Agent from Agoric Market institution 124
6.11 Augmented Agent 125
6.12 Fragment of GenericSeller Agent 126

7.1 Graphical Specification of the aucton house federation performative structure. 132
7.2 Specification of the auction room scene 136
7.3 Graphical specification of the conference centre performative structure. 141
7.4 Specification of the Appointment Proposal Scene. 143
7.5 Specification of a norm in the conference centre. 144

Foreword

Marc Esteva has managed to materialise the ideas of a very active research group within the IIIA during the last eight years. He has completed the formalisation of the Electronic Institution concept and has implemented a powerful tool called ISLANDER that helps engineers on the difficult process of specifying, verifying, and deploying multi-agent system applications. This tool has received several awards, among them the 'Best prototype paper' during the AAMAS02 Conference in Bologna. Also, the tool has been successfully applied to the development of a real application: a federation of concurrent fish auctions houses. It is a nice wrapping up for a period of research that started eight years ago with the Fish-Market idea materialised in Pablo Noriega's Ph. D. and that was fully developed in Juan Antonio Rodriguez's Ph. D. Multi-Agent systems development urgently requires well founded specification methodologies; during the last years we have witnessed the proposal of several such methodologies. In my opinion, Electronic Institutions, supported by the tool ISLANDER, are among the most solid ways of approaching the engineering of these systems. The reader will find in this book a detailed description of how to use the methodology together with complete examples and some initial steps towards a formal view of the underpinning ideas.

Marc's personality, always co-operative with his colleagues, has permitted the gathering of efforts, including several Master Thesis, around the development of the software that is presented in the book. His solid computer science background permitted him to drive the development without surprises, without errors, and steadily. I hope the reader enjoys the book and becomes an ISLANDER user as soon as ends the reading of it.

Bellaterra, October 2003

Carles Sierra
IIIA, CSIC

Abstract

This thesis focuses on the analysis, design and development of open multi agent systems. We argue that open multi agent can be designed and implemented as electronic institutions that define the rules that govern agent societies. Electronic institutions define what each agents are permitted and forbidden to do. At this aim, an institution defines the roles that participants can play, their valid interactions and their consequences. Furthermore, institutions enforce social and individual behaviour by obliguing everybody to act according to the norms. This thesis continues previous work on electronic institutions presented in Pablo Noriega's and Juan Antonio Rodríguez's PhDs.

Due to the complexity of open multi agent systems we advocate for a formal approach on their design and development. For this purpose we present a formalisation of electronic institution that permits a sound definition of all its components. The formalisation is the basis for the specification, verification and execution of institutions.

An important effort on this thesis has been devoted to the development of software tools that facilitate institution designers' work from the specification to the subsequent development of institutions. Firstly, we have defined a textual specification language for institutions based on the formalisation, the so-called ISLANDER, and more importantly, the ISLANDER editor, a tool for the specification and verification of institutions. On the one hand, it permits a complete specification of an institution allowing when possible to specify some of the elements graphically. Once specified an institution the tool is capable of verifying it, permitting to detect errors at an early stage.

We argue that the execution of an electronic institution requires an infrastructure that facilitates agents to participate in the institution while controlling that they behave according to the institution rules. Since open multi agent systems will be populated by heterogeneous and self-interested agents we can not assume that those agents will behave according to the institutional rules. Hence, we have developed a social layer middleware on top of a FIPA-compliant platform that facilitates participating agents the information they need to successfully participate within the institution, mediates agent interactions, and enforces the institutional rules to participating agents. The social layer middleware is generic in the sense that it can be used in the deployment of different institutions with no extra programming.

Acknowledgements

This thesis has been mainly supported by the doctoral CIRIT grant (1999FI-00012) and the eInstitutor project(TIC2000-1414).

First of all, I would like to thank Carles Sierra for accepting and supervising my PhD. This thesis would not have been possible without his support and advice. To work with him has been an invaluable experience.Furthermore,he has offered me the opportunity to collaborate with other researchers, as well as, to visit other research labs. Both activities have been extremely useful in my scientific career.

Parts of this thesis correspond to papers jointly elaborated with other researchers. I'd like to thank the co-authors of several of my papers that have authorised the use of parts of them as material to include in this dissertation monography.

The work on institutions' formalisation was done in collaboration with Juan Antonio Rodríguez, Pere Garcia and Josep Lluís Arcos. Special mention is devoted to Juan Antonio Rodriguez for his support, valuable advices and friendship during these years.

I want to thank the researchers working in the EU funded SLIE project for their interest in the work and software tools presented in this thesis. Although I was not directly involved in the project, I had the opportunity to collaborate with them being parts of this thesis the resukt of these collaborations. Concretely, the section on agents development in chapter 6 mainly corresponds to work and ideas developed by Wamberto Vasconcelos. The work on institutions' model checking was principally carried out by Mike Wooldridge, Marc-Philippe Huguet and Steve Phelps of the University of Liverpool.

I want to specially thank Julian Padget for hosting me at the University of Bath,as an ERASMUS student, and for giving me the possibility to write my first paper. The paper reported in chapter 4 would not have been possible without his help and in-depth knowledge on process algebras.

The software presented in this thesis would not have been possible without the work of Bruno Rosell and David de la Cruz. They have developed an important part of the software components presented here. I want also to thank Lucía Ramón for the grammar checking of an important part of this document.

The MASFIT project has been a perfect environment to use the ideas and software presented in this thesis. I want to specially mention the developers of the project, namely Guifre Cuní and Eloi Puertas for their patience and feedback. Since they where the first users of the software, they were its testers. They feedback and comments have been very valuable to improve it.

To work at the IIIA during these years has been a very exciting scientific and

personal experience. Apart from doing my PhD I have had the opportunity to meet very interesting people and to make a lot of friends. I want to thank all of them, staff researchers, PhD. students, administrative staff and visitors, for making the IIIA a nice place to work.

During my Phd. I have also had the possibility of visiting the "Laboratorio Nacional de Informática Avanzada" in Mexico and the Computer Science Department of the University of Lisbon. I want to thank the people in both places for their hospitality and friendship, specially Christian Lemaitre and Luis Antunes.

Finalment, els meus agraïments són per a tota la meva família i amics. Sense el seu suport i ànims aquesta tesi no hagués estat possible. A la meva mare i al meu pare per haver-me donat la vida, haver.me cuidat i sempre haver estat al meu costat. Al meu germà per ser el meu millor amic. A la meva àvia i a la meva tia per la seva estimacio i haver-me cuidat mentre escrivia la tesi.

Marc Esteva
IIIA CSIC
Bellaterra, 23 d' octubre de 2003

Chapter 1

Introduction

This thesis focuses on the analysis, design and development of open multi-agent systems. We argue that open multi-agent systems can be effectively designed and implemented as agent mediated electronic institutions where heterogeneous (human and software) agents can participate playing different roles and interacting by means of illocutions.

1.1 Motivation

Multi agent systems (MAS) are systems composed of autonomous agents which interact in order to satisfy the common and/or individual goals. MAS represent an appropriate solution for those problems in which knowledge, resources, and control, are partially and geographically distributed, and when some legacy systems must be made to network [Wooldridge and Jennings, 1999]. MAS technologies have been applied successfully to different areas, as for instance, air traffic control, electronic commerce, information management, etc (reviews of agent applications can be found in [Jennings and Wooldridge, 1998] and [Parunak, 2000]).

A main feature of MAS is that they communication occurs at knowledge level and that they use flexible and complex interactions among their components. Thus, the design and development of MAS have all the problems associated to the development of distributed, concurrent systems and the additional problems which arise of having flexible and complex interactions among autonomous entities [Jennings et al., 1998]. In order to cope with these problems appropriate methodologies which allow the analysis and design of agent systems and software tools which give support to all the stages of their development life cycle are needed [Jennings et al., 1998, Iglesias et al., 1999].

An important distinction when designing a multi agent system should be done between the macro-level (social) and micro-level (agent) aspects of the system. On the one hand, macro-level aspects focus on the social aspects of agents, that is, how the multi agent system should be structured defining the

relationships and interactions among the agents. On the other hand, micro level aspects focus on the internal aspects of agents, that is, which architecture the agents should have, and the reasoning mechanisms that agents should use to take their decisions. Early work in Distributed Artificial Intelligence identified the advantages of organisational structuring as a main issue to cope with the complexity of designing DAI systems [Gasser et al., 1987, Pattison et al., 1987, Corkill and Lesser, 1983, Werner, 1987]. Organizational approaches address the problem of designing a MAS from a macro level perspective defining the roles that participating agents may play and structuring the valid interactions that they may have. Nonetheless, agent researchers have spent more time and effort defining and developing agent architectures theories and languages, than in defining general methodologies which allow to formalise agent systems from their macro level perspective and which should give support to all the stages of their development [Iglesias et al., 1999].

The complexity on designing multi agent systems increases when we focus on open systems [Hewitt, 1986]. Open multi agent systems are those in which the participants are unknown in advance and can change over time. These systems are populated by heterogeneous agents, normally developed by different people using different languages and architectures, representing different parties and acting to maximise their own utility. Furthermore, participants in open multi agent systems change over time, new agents can appear and some of the participants can disappear. Obviously, this type of systems must be designed taking a macro level view of the system as all the agents are not known at designing time. Open multi agent systems represent arguably the most important area of application of multi agent systems [Wooldridge et al., 1999]. The goal of the methodologies for open multi agent systems should not be to obtain a set of agents which interact following the defined interaction protocols to achieve a common objective, it should be to define a normative environment which will determine agent possible behaviours within the system. In other words, the methodologies for agent systems should focus on identifing the different roles that agents can play within the system, the relationships among them and structure their possible interactions. Thus, methodologies which commit to a concrete agent architectures or focus on the internal aspects of the agents, as for instance Tropos [Giunchiglia et al., 2002] or Prometheus [Padgham and Winikoff, 2002], are not appropriate for open multi agent systems.

Human societies have coped with similar problems to those associated to open multi agent systems by following conventions and deploying institutions. Human interactions very often follow conventions, that is, general agreements on language, meaning, and behaviour. By following conventions humans decrease uncertainties about the behaviour of others, reduce conflicts of meaning, create expectations about the outcome of the interaction and simplify the decision process by restricting to a limited set the potential actions that may be taken. These benefits explain why conventions have been so widely used in many aspects of human interaction: trade, law, games, etc.

On some occasions, conventions become foundational and, more importantly,

some of them become norms. They establish how interactions of a certain sort will and must be structured within an organization. These conventions, or norms, become therefore the essence of what is understood as human institutions [North, 1990]. This is so for instance in the case of auction houses, courts, parliaments or the stock exchange. Human institutions not only structure human interactions but also enforce individual and social behaviour by obliging everybody to act according to the norms.

The benefits obtained in human organizations by following conventions become even more apparent when we move into an electronic world where human interactions are mediated by computer programs, or agents. Conventions seem necessary to avoid conflicts in meaning, to structure interaction protocols, and to limit the action repertoire in a setting where the acting components, the agents, are endowed with limited rationality. The notion of electronic institution becomes thus a natural extension of human institutions by permitting not only humans but also autonomous agents to interact with one another. Thus, electronic institutions are thought to define the rules of the game in agents societies.

Due to the nature of multi-agents systems composed by autonomous and distributed entities which interact an important part of the effort in its development is spent on building basic communication and coordination mechanisms [Jennings et al., 1998]. Since most of this communication and coordination mechanisms are domain independent, they can be grouped together in generic infrastructures that can be used to deploy multiple MAS and avoiding to develop each agent based system from scratch. Hence, the development of generic infrastructures which allows the rapid development of multi-agent applications has been identified as a main issue for the succeed and expansion of multi-agent technologies [Ashri and Luck, 2001]. The development of generic infrastructures intends to facilitate the work of system and agent designers allowing them to primarily focus on domain dependent issues and on agent decision making mechanisms. The resulting infrastructure should ideally provide participating agents with the services they need in order to successfully fulfil their goals. Furthermore, in the case of open multi agent systems as they will be populated by self interested agents, they should be composed by the agents taking part in the system and the computational mechanisms realising the rules of the society [Rodríguez-Aguilar, 2001]. In other words, the computational mechanisms which enforce the rules of the society to participating agents or which detect agents violations of them.

1.2 Contributions

This thesis is a continuation of the work on electronic institutions already presented in [Noriega, 1997, Rodríguez-Aguilar, 2001]. The idea of agent mediated electronic institution, as the electronic counterpart of human institutions, was introduced in [Noriega, 1997]. Using as a motivation example a typical trading institution, fish market auction houses, Noriega introduces the different components of an institution. He proposes that an institution is defined by a set

of roles and relationships among them, a common ontology and communication language which allow heterogeneous agents to exchange knowledge, the valid interactions that agent may have structured in conversations, and a set of rules of behaviour which determine the actions that agents must do under certain circumstances.

The formalisation of institutions presented by Noriega was extended and refined in Rodríguez's thesis [Rodríguez-Aguilar, 2001]. Furthermore, an important effort on Rodriguez's thesis is devoted on the realisation of electronic institutions. He discusses how electronic institutions can be realised and as a proof of concept presents a detailed description of the fish market implementation.

Taking into account their previous work the objectives at the beginning of this thesis were: to continue the work on the institutions' formalisation; to give support to institution specifications and their automatic verification; and the development of a generic infrastructure which could be used for the deployment of the specified institutions. Chapter 3 focuses on the institution formalisation, chapter 5 on their specification and verification, and chapter 6 focuses on the realisation of institutions.

Due to the complexity of designing electronic institutions, we advocate for a formal approach which should guide their analysis, design and development. For this purpose and continuing the previous work on electronic institutions [Noriega, 1997, Rodríguez-Aguilar, 2001, Esteva et al., 2001], we present, in chapter 3, a formalisation of electronic institutions where we have refined and extended some of their components. From our point of view an electronic institution is defined by a set of roles, a common language, the activities that can be done within the institution and the consequences of agents' actions within the institution. First of all, each role defines a pattern of behaviour within an institution and allows us to abstract from the concrete agents that will populate the institution at execution time. As actions are associated to roles, what a participating agent may do within an institution is determined by the roles that it can play. Obviously, in order to allow heterogeneous agents to exchange knowledge, a common language and ontology must be defined which are used to define agent interactions. Agent activities within the institution are structured in *conversations* which determine the valid interaction that agents may have and represent the context where exchanged illocutions must be interpreted. Furthermore, the institution defines the valid conversations that agents may have and how depending on their roles they can move among them including the definition of when new conversations can be created. Finally, the institution defines the consequences of agents' actions within the different conversations which can either limit or expand future agent acting possibilities and can impose *obligations* to participating agents.

Taking as a basis the institution formalisation we advocate that the development of electronic institutions must be preceded by a formal specification which permits to identify and formalise all their components. For this purpose, we have defined a textual specification language, the so called ISLANDER, based

on the institutions formalisation, and more important we have developed the ISLANDER editor, a tool for the specification and verification of institutions. The language and the tool are both presented in chapter 5. The ISLANDER editor permits the complete specification of an electronic institution allowing, whenever possible, to specify elements graphically. We believe that graphical specifications are extremely useful as they facilitate the designer work and they are also much easier to understand. Outputs of the tool are the specified institution on the defined textual specification language and on XML. Moreover, the tool also verifies the correctness of specifications. For instance, that within a conversation protocol a final state is always reachable.

The institution formalisation and the specification language focus on the macro-level aspects of the system and not on micro-level (internal) aspects of participating agents. Since no assumptions are made and no restrictions are imposed on the internal characteristics of participating agents, the specified institutions are architecturally neutral and agent designers can choose the language and architecture which better fits their goals.

Notice that we take the view that *all* interactions among agents are carried out by means of *message interchanges*. Thus, we take a strong dialogical stance in the sense that we understand a multi-agent system as a type of *dialogical system*. The interaction between agents within an institution becomes an illocution exchange. In accordance with the classic understanding of illocutions (e.g. [Austin, 1962] or [Searle, 1969]), illocutions are not simply propositions that are true or false, but attempts on the part of the speaker that succeed or fail.

The institution specification defines a normative environment which restricts agents possible behaviours structuring their interactions and defining what agents are permitted and forbidden to do depending on their roles. Since electronic institutions, and open multi agent systems in general, will be populated by heterogeneous and self interested agents representing different parties, it can not be assumed that these agents will behave according to the institutional rules. For this reason, we take the view that agents participation within an institution should be mediated by an infrastructure which provides agents with the information they require to successfully participate in the institution, facilitates agents communication with other agents within the different conversations, keeps track of each agent pending obligations and guarantee the correct evolution of institution execution by enforcing the institutional rules encoded in the specification.

Instead of developing an infrastructure from scratch we have opted for developing a social layer middleware on top of the JADE platform [Bellifemine et al., 2001] which we use as a communication layer among agents. The social layer is thought to cope with the institution concepts at execution time. That is to say, the agents of the social layer control the roles that participating agents are playing, the current conversations that are taking place within the institution, and keep track of each agent pending obligations. Furthermore, they coordinate to guarantee the correct evolution of each conversation and agent movements among them, and verify that agents' actions within the institution are correct with respect to the institution specification and the current execu-

tion. The developed social layer is generic in the sense that it can be used for the deployment of different institutions as the agents composing it are capable of loading institution specification as generated by the ISLANDER editor. The infrastructure architecture, the different agents that compose the social layer and how they manage institution concepts at execution time are presented in chapter 6.

Furthermore, the last part of the chapter is devoted to show how agents for electronic institutions can be developed. Since institution specifications does not contain information about how agent have to take their decisions, they can not be automatically generated from institution specifications. However, we show how agent skeletons can be generated from the institution specifications. Skeletons are basic programs with the capacity to navigate among the different institution conversations. In order to develop the agents engineers should customise the skeletons by selecting the roles that the agent is going to play, the conversations in which it will participate and defining the agent decision making mechanisms [Vasconcelos et al., 2002a, Vasconcelos et al., 2002b, Vasconcelos et al., 2003].

Process algebras is a research area focussed on the formal analysis of concurrent and distributed systems. Thus, they can be used to specify, analyse and verify multi agent systems from a distributed point of view. In chapter 4, we present an alternative architecture for agents in an auction and a distributed mechanism for the bidding resolution process, formalised and implemented using π-calculus [Esteva and Padget, 2000]. For a formalisation of electronic institutions in ambient calculus and sale calculus reader is referred to in [Esteva et al., 2002b] and [Padget, 2001].

In order to illustrate how electronic institutions can be practically specified we present two examples on chapter 7, an auction house federation and a conference centre. The auction house federation is an institution which permits buyer agents to participate in real fish markets in the same conditions as human buyers. Several fish markets can be connected to the federation allowing buyer agents to receive information from all of them and decide which is the better place to buy. For this purpose, the institution is connected to the software system in charge of each real auction house from which it receives information of the events occurring in the auction house and to which it sends the bids submitted by buyer agents. The auction house federation is a continuation of the fish market institution already presented in [Noriega, 1997, Rodríguez-Aguilar, 2001].

The second example corresponds to the conference centre institution [Arcos and Plaza, 2002]. Complementary to the physical space where the conference takes place it is defined an institution where agents representing the conference attendees interact in order to look for interesting activities for them. Each attendee can customise different Personal Representative Agents (PRA) which participate in the institution on behalf of it. PRAs devote their time within the institution to look for interesting events or activities for their attendees and negotiating appointments with other PRAs representing attendees with similar interests.

1.3 Strucutre

This thesis is divided in eight chapters including this one and one appendix:

Chapter 2 presents an overview among the research areas relevant for this thesis. The chapter is divided in an overview on agent oriented methodologies, design mechanisms of agent interactions and agent platforms.

Chapter 3 focuses on the formal model for electronic institutions. It presents the different components of our electronic institution model and a formal definition of each one of them.

Chapter 4 presents a formalisation of an alternative architecture for an auction room and a distributed mechanism for the bidding resolution process. The system has been formalised and implemented using π-calculus.

Chapter 5 describes the ISLANDER specification language and the ISLANDER editor. ISLANDER is a textual specification language for electronic institutions that we have defined based on the institutions formalisation presented in chapter 3. Furthermore, we have developed an ISLANDER editor, a tool for the specification and verification of electronic institutions. We describe the ISLANDER editor and the verifications that it does on the specified institutions.

Chapter 6 focuses on the execution of electronic institution. We propose a multi-layered architecture for electronic institutions composed by a communication layer, a social layer and an autonomous agent layer composed by the agents taking part in the institution. Principally the chapter is devoted to explain the agents composing the social layer and how they handle institution concepts at execution time. Furthermore, the last part of the chapter focuses on how agents for electronic institutions can be developed.

Chapter 7 presents two examples of electronic institutions: the auction house federation and the conference centre institutions.

Chapter 8 outlines the conclusions of this thesis and future research.

Appendix A presents the complete specification of the institution examples presented in chapter 7 in the ISLANDER language.

Chapter 2

State of the Art

Distributed Artificial Intelligence (DAI) is a subfield of Artificial Intelligence concerned with distributing and coordinating knowledge and actions. DAI systems consist on several entities with certain degree of autonomy and intelligence which interact to achieve their common and/or independent goals.

There is no agreement on a concrete agent definition but here we adhere to the definition proposed in [Jennings et al., 1998]: "an agent is a computer system, *situated* in some environment, that is capable of *flexible autonomous* action in order to meet its design objectives". *Situated* means that it can receive inputs from its environment and it can perform actions that change the environment. *Autonomy* means that the system should be able to act without the direct intervention of humans and it should have control over its own actions and internal state. Finally *flexible* means that the system is *responsive*, *pro-active* and *social*, where: *responsive* means that agents should respond in a timely fashion to changes that occur in their environment; *pro-active* means that they should be able to take the initiative and exhibit a goal-directed behaviour; and *social* means that agents should be able to interact.

Next we present a short description of the state of the art in the areas related to the thesis: Multi-Agent Systems, Agent Communication and Agent Platforms.

2.1 Multi-agent Systems

Multi-agent systems(MAS) are systems composed by several autonomous agents which interact in order to achieve some goals. One of the main characteristics of MAS is the use of sophisticated interaction patterns, as for instance [Jennings et al., 1998]: cooperation (working together towards a common aim); coordination (organising problem solving activity so that harmful interactions are avoided or beneficial interactions are exploited); and negotiation (coming to an agreement which is acceptable to all the parties involved). It is the flexibility and high-level nature of these interactions which distinguishes multi-agent systems from other forms of software and which provides an underlying power of

the paradigm.

Agent-based systems are adequate for those problems where data, control, expertise or resources are distributed; agents are a natural approach for delivering system functionality; and where a number of legacy systems must be made to network [Wooldridge and Jennings, 1999]. An important feature of an agent-based approach is that it can incorporate legacy software systems. This can be done by wrapping legacy systems by an agent layer in charge of communicating it with the other entities or by a transducer, a program which receives the messages addressed to the legacy system and translates it in a format which can be understood by it and realises the inverse operation when the legacy system wants to send a message through the network [Genesereth and Ketchpel, 1994]. Another important advantage of agent-based approaches is reusability of agent architectures and interaction protocols in different applications [Jennings, 2000]. For instance, agents with BDI architecture and general interaction protocols as for instance, resource allocations protocols as Vickrey or English protocols, have been used in many applications.

The complexity of designing and developing MAS arise from their distributed nature and from having high level and flexible interactions among autonomous entities [Jennings et al., 1998]. Then, appropriate methodologies which permit the analysis and formalisation of the system taking into account agent characteristics and which give support to all the phases of their development are needed [Iglesias et al., 1999]. Furthermore, the complexity inherent to multi agent systems increases when we consider open multi-agents systems. These are systems whose components are not known in advance and they can change over time. Open MAS are populated by heterogeneous and self interested agents, which represent different parties and act to maximise their own utilities. The former means that open MAS should accept agents developed using different languages and architectures. The second means that open MAS should implement the mechanism to react and protect from agents deviating and fraudulent behaviours, allowing agents only to perform the actions which are authorised for [Dellarocas and Klein, 1999, Rodríguez-Aguilar, 2001].

The expansion of Internet with an exponential increase on the number of companies and people who has access to the networks demands for new applications which allow them to interact in order to satisfy their goals. We believe that this represent a challenge for MAS technologies which can handle with the requirements of this type of applications. Hence, open MAS represent arguably the most important application area of multi-agent systems [Wooldridge et al., 1999]. A typical example of open multi-agent systems are marketplaces over Internet where agents on behalf of users sell and buy products. Nonetheless, the success of MAS technologies and their expansion depends on the existence of appropriate methodologies and software tools which facilitate their design and rapid development.

In order to cope with the complexity of designing MAS systems the idea of modelling multi agent systems as organisations was early proposed [Gasser et al., 1987, Pattison et al., 1987, Corkill and Lesser, 1983,

Werner, 1987]. Organizational approaches propose to analyse MAS from a global perspective structuring the agent society by identifying the roles that participating agents may play and the relationships among them. They consider organizations as first class citizens of MAS and defend that any agent interaction occurs in the context of an organisation.

However, organisational approaches have not been a common use in MAS where most systems have been designed without analysing the system as a whole and taking an agent-centred view. In these latter cases, the modelling of the system consists on identifying the different agents, assigning the tasks to them and defining the set of interactions that they need in order to accomplish the tasks. They usually see a multi agent system as a pure aggregation of agents. Furthermore, in many of the systems following an agent-centred approach there are strong assumptions of agents' benevolence and cooperation which can not be taken in open systems. Agent-centred approaches can be useful for closed systems composed of a small number of agents but they fail to design open systems [Rodríguez-Aguilar, 2001, Esteva et al., 2001].

Different methodologies and approaches has been proposed for the design and development of MAS. Extended revisions of existing agent-oriented methodologies can be found in [Iglesias et al., 1999, Wooldridge and P.Ciancarini, 2001]. We can see that most of them are extensions of existing methodologies in other fields to include relevant aspects of agents. These extensions have been carried out mainly from two areas: object oriented and knowledge engineering. But these approaches do not conveniently cover all the aspects needed on the design of MAS. As it is defended in both revisions, we believe that it is necessary to develop specific methodologies to cope with the complexity of MAS design and development. Next we describe some of the methodologies and approaches proposed for MAS.

Gaia [Wooldridge et al., 2000, Wooldridge et al., 1999] is a methodology for the analysis and design of multi-agent systems as computational organisations consisting on various interacting roles. At analysis stage the role and interaction models are defined. Each role is defined by four attributes: responsibilities, permissions, activities and protocols. Responsibilities determine the functionality of the role which can be divided in liveness and safety responsibilities. Liveness responsibilities are expressed as regular expressions over the role activities and interactions, and define the role life-cycle. Safety responsibilities express conditions that a role must preserve and are specified as predicates over variables in the role permissions. Permissions determine the resources available for a role in order to realise its responsibilities while activities represent private agent actions carried out without interaction with other agents. Complementary, the interaction model defines the protocols among roles capturing role dependencies and relationships. Protocols are specified at high level defining the purpose of the interaction but not the concrete sequence of messages. Thus, each role has also a set of protocols associated which determine how it can interact with other roles.

At design stage the role and interaction models are used to define the agent

model, the service model and the acquitance model. The agent model defines the agent types that will compose the system and the cardinality of each one. Concretely, agent types are defined by assigning them one or more roles. The service model defines the services associated to agent types where a service is a coherent block of functionality and derives from the activities and protocols of the roles. Finally, the acquitance model defines the communication links among agent types. The objective of the design stage in Gaia is to reach a level of detail that can be directly implemented by engineers. Major criticisms to Gaia methodology is that the organisational structure is only implicitly (not explicitly) defined and that it is not defined for designing open systems as there is an assumption of a common goal among agents.

The ROADMAP methodology [Juan et al., 2002] extends the Gaia methodology for the analysis and design of complex open systems. As a first extension a use case model is introduced to support requirements gathering. From the use case model an environment and knowledge models, not present in Gaia, are derived defining the execution environment and domain knowledge respectively. The interaction model in Gaia is renamed to the protocol model and a new interaction model is added which permits to specify the protocols in AUML. The role model is extended with a role hierarchy represented as a tree where the leafs are atomic roles defined as in Gaia, and the rest of the roles are defined as an aggregation of its sub-roles. An interesting feature of the extended role model is that it permits runtime reflection allowing to change the role model. That is, a role can have permissions to change other role attributes.

A totally organisational approach is taken in [Ferber and Gutknetch, 1998] where organisations are defined based on three main concepts: group, role and agent. In order to abstract from the concrete agents that will compose the system, each group is defined by the set of roles that can participate in it and the interaction protocols that these roles may have specified as role-to-role protocols. Agents within a group must play some of its roles. Furthermore, each agent can participate in different groups at the same time. In order to validate the approach the MadKit platform has been developed permitting the design and execution of MAS organisations.

The Tropos methodology [Giunchiglia et al., 2002] covers the overall software development process from the early requirements to their implementation. They divide the construction of a MAS system in five phases: early requirements, later requirements, architectural design, detailed design and implementation. In the early requirements phase the relevant stakeholders, represented as actors, and their goals are identified where an actor represents an agent (software or physical), a role, or a group of roles, normally played by one agent. The system to be developed is added as another actor in the later requirements phase and its relationships with the environment are represented as dependencies among the system and the actors identified in the previous phase. In the architectural design phase the system is decomposed by adding new actors to which sub-goals and sub-tasks of the system to develop are assigned. Next, in the detailed design phase the systems'actors are defined in detail and the coordination and commu-

nication protocols using AUML are specified . At the implementation phase the specification produced at the detailed design phase is transformed into agent skeletons using a BDI architecture which are extended to completely develop the agents. Concretely, skeletons for the JACK platform [Howden et al., 2001] are obtained.

Prometheus [Padgham and Winikoff, 2002] is a methodology for specifying, designing, and implementing multi agent systems. The methodology consists of three phases: system specification, architectural design, and detailed design. In the system specification phase the functionalities of the system are identified. For each functionality, its actions, percepts (inputs), the data to which it has access (write or read) and the other functionalities with which it interacts are defined. The architectural design phase is devoted to decide which will be the agents of the system taking into account the different functionalities identified in the previous phase. A set of functionalities are assigned to each agent and the interaction protocols among them are specified in AUML. The detailed design phase focuses on developing the internal structure of the agents using a BDI architecture, taking into account the functionalities assigned to each agent. In order to facilitate the work of system engineers a software tool which gives support to the design of the system and generates agents skeletons for the JACK platform which can then be then customised has been developed. This methodology has been used successfully for the development of closed systems but results inappropriate for open systems.

In [Dignum, 2002] it is proposed that institutions' objectives, the values which lead to the fulfilment of these objectives and institution norms should be defined in a more abstract level than institution structures and procedures. At this abstract level general regulations and laws can be incorporated into the institution norms. For instance, a trading institution must respect trading laws in the country where the trading takes place. They propose the use of deontic logic for defining the abstract norms. In order to apply the norms to a concrete institution they have to be translated into concrete norms expressed in similar terms to which institution structures and procedures are specified. Finally, concrete norms are translated to rules that either specify the part of the procedures that enforce them or specify the events and triggers that signal a violation. Based on these ideas, HARMON*IA* [Vázquez-Salceda and Dignum, 2003] a framework for the design of agent organisations has been defined.

As we have said most of the researchers have tried to adapt object-oriented methodologies to design multi-agent systems including the relevant aspects of agent systems. As Unified Modelling Language (UML) became a *de facto* standard of object oriented modelling, many researchers have tried to adapt their notation, diagrams, and models to agent systems [Odell et al., 2000, Parunak and Odell, 2002, Bauer et al., 2001]. For those engineers familiar with UML diagrams and tools, AUML represents a natural extension when moving from the object oriented world to the agent world. Notice that UML, and as a consequence AUML, is not a methodology, it is a language to define the system. Different methodologies propose the use of extended UML diagrams and mod-

els to define different components. For instance, ROADMAP proposes the use of use cases to capture the requirement of the system and AUML interaction diagrams to represent interactions among agents.

In [Dellarocas and Klein, 1999] they advocate to differentiate between the social design and agents' design, that is between the rules that govern the society and the participants, the agents. Furthermore, they argue that it can not be expected that participating agents in open MAS will behave according to the rules. Then, they propose that agent societies have to contain several institutions, namely socialisation service, notary service and exception handling service. The socialisation service is responsible to authorise agents to participate in the society. The result of this process is a contract that gives participating agents the authorisation to participate in the society identifying their rights and capabilities. The notary service is in charge of verifying that agent interactions are legal with respect to the rules of the society. Finally, the exception handling service is in charge of anticipating, avoiding, detecting, and resolving all known exception types. This is done by "sentinel" agents.

Obviously, we have to make reference to the previous work in electronic institutions [Noriega, 1997, Rodríguez-Aguilar, 2001] which constitute the starting point of this thesis. In [Noriega, 1997] the notion of agent mediated electronic institution was introduced. Using as a metaphor, a traditional trading institution, as the fish market, it defines the basic concepts that compose an electronic institution. In [Rodríguez-Aguilar, 2001] the ideas already presented are extended and refined. Furthermore, an implementation of the fish market was presented and a proposal of how the infrastructure used in it can be extended to be general enough to be used in the deployment of any intitution.

Continuing their work we refine and extend the electronic institutions formalisation, we present a textual specification language, a specification and verification tool and a generic infrastructure which can be used in the deployment of different institutions.

Analysing the different methodologies we can see that, except for Prometheus, they consider the notion of role as fundamental in order to formalise MAS. Nonetheless, the functionalities identified in Prometheus can be seen as roles. Thus, a fundamental aspect of any methodology is to identify the roles and their relationships. In ROADMAP, Tropos, and Prometheus, protocols are specified using AUML while we have opted for using finite state machines to model conversations. But probably the main difference among them is the kind of systems that each one can model and the final result of each methodology. The objective of Tropos and Prometheus is to obtain a set of agents which behave to achieve the different goals of the system and interacting when necessary following the defined protocols. They represent a perfect example of designing and developing a multi agent system taking an agent-centred approach. There is no notion of organisation, they commit to a concrete agent architecture and there is an assumption of agents benevolence and cooperation among the resulting agents. Although they can be useful on the design of closed systems, and they offer different tools which generate agent skeletons reducing the development

time of the system, they are not appropriate for open systems.

The other approaches, as in our case, do not commit to any agent architecture assuming that the system will be populated by heterogeneous agents. Furthermore, in Gaia the organisational structure is only implicitly defined and the final result of the methodology is a level of detail from which the system can be implemented and interaction protocols are only described but not specified. We believe that the result of the design stage of methodologies for open systems should be a complete and sound definition of the rules that govern the organization. In other words, the definition of a normative environment which restrict agent possible behaviours. For this purpose, similarly to [Ferber and Gutknetch, 1998] we take a totally organisational approach by defining the structural organisation of the system and without making any assumption about the internal characteristics of the participating agents.

We see the work on abstract norms of [Dignum, 2002] at a higher level of abstraction with respect to our model of institutions. Furthermore, they advocate to transalate the abstract norms to a concrete norms expressed in the same language of institutions strcuctures and procedures. We believe that concrete norms can be used for checking if a designed institution implements the abstract norms or if behaviours which violate the norms are possible within it.

Finally as [Dellarocas and Klein, 1999] we advocate that open multi agent systems must contain computational mechanisms which are in charge of controlling that participating agents do not violate the rules governing the society at execution time. That is, the final result of methodologies for open systems must include the computer mechanism which imposes the normative environment to participating agents. For this purpose, we propose an infrastructure which facilitates agents participation within the institution but enforcing the institutional rules encoded in the system specification.

2.2 Agent Communication

Since interaction is one of the basis of multi-agent systems much of the effort in multi-agent research has focused on agent communication. In order to allow agents to communicate, the first step is to define a common language. Different Agent Communication Languages (ACL) have been proposed based on speech act theory [Austin, 1962, Searle, 1969]. Speech act theory is based on the observation that utterances must not be considered as simple propositions, but as speaker's attempts that succeed or fail [Austin, 1962].

ACLs have been defined to allow agents to exchange information and knowledge, and what distinguishes them from other ways of communication are the objects of discourse and their semantic complexity [Labrou et al., 1999]. ACLs address communication at the intentional and social level [Dignum and Greaves, 2000]. Generally an ACL is composed of three main elements [Genesereth and Ketchpel, 1994]: a vocabulary, an inner-language to encode the knowledge to be exchanged among agents using the vocabulary offered by the ontology and an outer language to express agents'

intentions. The vocabulary should be contained in ontologies shared by the different agents engaged in an interaction. Examples of ACL's are: FIPA ACL [FIPA, 1997] and KQML [Finin et al., 1995]. But agents normally do not engage in a single message exchange, similarly to human beings, they engage in conversations which define the valid sequences of exchanged messages and define the context in which exchanged messages must be interpreted [Labrou et al., 1999]. Hence, the notion of *conversation* as the unit of communication among agents has been promoted [Greaves et al., 2000].

An important decision when specifying conversations is to choose between specifying the global protocol or the different agents view of the protocol. In the first case the protocol is specified containing all the messages that agents can exchange and the definition of the protocol contains information about who can send the messages and to who can be addressed. If the conversation is specified from an agent view point, the specification of the protocol contains only the messages that an agent involved within the conversation send or receive. In this case, a different vision of the protocol should be specified for each type of agent that can be engaged in the conversation which corresponds to their view of the conversation. Then, each agent engaged in the conversation manages a different version of the protocol.

Assuming conversation as the unit of communication the next issue is deciding how conversations have to be specified. Agent researchers have mainly opted for specifying conversations using finite state machines (FSM), Petri nets, or extensions of them.

Finite state machines offer a really intuitive way to specify conversation protocols. Normally, the different states of the FSM represent the different states of the conversation and the arcs connecting the states are labelled with the actions that make the conversation evolve from the source state of the arc to its target state. Thus, FSM and different extensions of them, have been widely used for specifying agent interactions. The main criticisms to the use of FSM is that they are not adequate to express concurrency. In other words, they are adequate for specifying sequential interactions but fail to model conversations where concurrency is allowed. Due to this reason the use of Petri nets and extensions of them has been promoted for specifying agent conversations because:

- They have a graphical representation;
- They support for concurrency;
- They are well researched and understood and have been applied to many real word applications; and
- There are many tools for the design and analysis of CPN-based systems.

On the contrary, the specification of protocols using Petri Nets is not as easy as in the case of FSM approaches. Moreover, the major criticism to the use of Petri nets is the combinatorial explosion on the size of the network as the complexity of the protocol increases.

Since FSM and Petri net approaches have been the more common used for modelling agent interactions, in the last years, an important number of researchers are opting for using AUML to specify agent interactions. Then, in [Odell et al., 2000, Odell et al., 2000] it is reported how UML diagrams can be extended and used to model agent interactions. As we have pointed out in the previous section some of the methodologies have opted to specify several aspects of MAS design by using AUML. The motivations for using AUML to specify agent interactions are that they have an intuitive graphical representation, are familiar to those researchers coming from the object oriented world and the existence of tools which permit their representation. The major criticism is that they are not based on a formal model which permits their formal verification. Hence, their verification requires to translate them to another formalism.

One of the main approaches using FSM is COOL [Barbuceanu and Fox, 1995]. Conversation protocols in COOL are specified from the agent view point by using a FSM where the states represent the different states of the conversation and arcs are labelled with the utterance or reception of a speech act. Each agent participating in a conversation manages a different version of the protocol. For each type of conversation that agents may have, a conversation class is defined which includes: the definition of the protocol; conversation rules which define what agents have to do when they receive a message; and error rules which define how to recover from unexpected events, as for instance, the reception of an unexpected message. Furthermore, each agent has a set of continuation rules which define whether an agent will accept requests to start new conversations or select one of the current conversations to continue. An important feature of the system is that ongoing conversations can be suspended to start a new conversation. When the new conversation is finished the suspended conversation is resumed.

In [Nodine and Unruh, 1999] finite state machines are used to define conversations between two agents: an initiator which is the agent that starts the conversation and a responder, within the InfoSleuth agent system. They also present how conversations can be extended and concatenated. The extension of a conversation consists on expanding a conversation to handle unexpected events. This permits agents to discard a conversation at a intermediary state or to handle erroneous messages with respect to the protocol definition. Extensions are specified independently of the conversation and can be applied to some of them. The concatenation of two conversations consists on connecting the final state of one conversation to the initial state of the other. In order to enforce the conversation, they have introduced a conversation layer into the agents generic shell, which is in charge of guaranteeing the correct evolution of the conversation within the local agent and the remote agent.

In Agentis [d'Inverno et al., 1998] protocols are specified as FSMs and are associated to services and tasks. There is a set of protocols that agents can instantiate depending on their goals. Once the specification is finished, protocols are translated to Z language [Diller, 1990]. This allows the use of tools developed for the verification and animation of Z specifications to verify and animate the

protocols.

In [Martín et al., 2000] it is proposed the use of Pushdown Transducers (PDT) a FSM extended with two tapes to specify and manage conversation protocols. The input is given by a pair of symbols and the stack is used to store and subsequently retrieve the context of the ongoing conversations. They advocate for the use of a special type of agent, the so-called *interagent*, devoted to guarantee that agents follow the specified protocol. Concretely, each agent engaged in a conversation is connected to an interagent which mediates its communication with the rest of the agents. Protocols are specified from the point of view of participating agents and interagents of agents engaged in the same conversation manage complementary conversations protocols. The main limitation of their approach is that the presented version of interagents can only manage conversation between two agents.

In the Java-based Agent Framework [Chauhan, 1997, Galan, 2000] the different agent views of the protocols are specified using FSMs and then, translated into a Petri net. Existing tools in Petri net are used to verify safeness, liveness and no dead lock properties for the specified conversation protocol. Concretely, they have developed a software environment which permits the specification of the protocols and which executes a Petri Net tool in order to verify the protocol.

In [Koning et al., 1998] it is proposed the use of Petri nets to specify and verify agent interactions. Their proposal consists on building a Petri sub net for each agent taking part in the conversation from a state transition diagram. Each Petri sub net models an agent's view point of the conversation. The different Petri sub nets are mixed to build the general Petri net for the conversation which is used to verify the protocol. This process is done by using a Petri net simulation and verification tool which allow for the graphical specification of the Petri nets and their subsequent verification by the simulation of the specified Petri nets.

The use of a particular type of Petri nets, hierarchical Colour Petri Nets, to specify agent interactions in which concurrency is needed is proposed in [Cost et al., 1999]. In order to specify the conversations they use the DesignCPN modelling tool.

In [Sibertin et al., 2000] it is proposed that agent interactions should be mediated by a *Moderator* which is in charge of guaranteeing the correct evolution of the protocol. Moderators are specified, validated, and implemented using CoOperative Objects [Sibertin-Blanc, 2001] a High-Level Petri Net language. Each CoOperative Object instance is a process which executes the multi-threaded behaviour defined by its Petri net. When a conversation is created, a moderator for it is launched and each agent wanting to perform a speech act sends it to the moderator who verifies that the action is correct with respect to the protocol definition. The moderator has a thread for each agent taking part in the conversation in order to keep track of its state within the conversation. Finally, when the conversation finishes the moderator disappears.

In [Mazoui et al., 2002] protocols are specified in AUML [Odell et al., 2000] and then translated into Coloured Petri Nets (CPN). The generated Petri net

contains one sub-net for each role taking part in the system and can be used to verify different properties of the protocol. For instance, that the protocol is deadlock free, that all the operations are executable, and that final transitions are always reachable form any mark accessible from the initial one. They also propose the study of interaction executions among a group of agents. For this purpose traces of agent interactions are stored which are then used to construct a global causal graph containing all the events produced during the system execution. The global causal graph is analysed in order to discover which of the specified protocols, stored in a CPN library, have occurred among agents and to analyse each one of them.

Since we do not allow concurrency within a conversation we have opted for specifying agent conversations using a FSM model and for specifying the global protocol of the conversation. Hence, some properties of the specified protocol can be verified by adapting search graph algorithms. Furthermore, we have explored how model checking can be used to verify some of the properties. We advocate for the enforcement of conversations to participating agents as in [Nodine and Unruh, 1999, Martín et al., 2000, Sibertin et al., 2000]. In this sense we advocate for a similar approach to the one used in [Martín et al., 2000], instead of incorporating a communication layer to agents as it is done in [Nodine and Unruh, 1999] or to have a single entity in charge of controlling the conversation evolution as it is done in [Sibertin et al., 2000]. In order to guarantee the correct evolution of a conversation each agent engaged in it, is connected a special type of mediator agent which we call governor [1] in charge of guaranteeing that agents engaged in a conversation follow the specified protocol.

2.3 Agent Platforms

Although the use of a methodology permits the analysis, specification and verification of the system the process that leads from the system specification to the real implementation is still a hard task. In this sense the development of generic infrastructures and software tools which give suport to agent based systems development are needed.

The development of multi agent systems is a hard and difficult task as it includes the development of each agent internal behaviour and the communication and coordination mechanisms needed to allow agents to interact. On the one hand, for each agent it should be implemented their behaviour, that is the computer mechanisms in charge of deciding the actions that the agent should do in each moment in order to satisfy its goals and how the agent should react to external events, as for instance the reception of a message from other agents or a change in the external environment. Complementary, it should be implemented the computer mechanisms that allow agents to communicate and coordinate, covering from low level communication issues, as providing a reliable transport service, to the high level issues, as for instance, the coordination mechanisms

[1] Reader is referred to section 6.1 for an extended discussion about the differences between interagents and governors in the management of conversations

to ensure the correct evolution of conversation protocols to which agents are engaged.

Since some of the communication and coordination mechanisms are domain independent they can be grouped together in generic infrastructures which can be used in the deployment of multiple MAS. Unfortunately, this has not been the case and most of the agent aplications have developed their own infrastructure with the consequence that an important part of its development time is spent in building basic communication and coordination blocks, and that they can not be reused [Jennings et al., 1998]. Furthermore, another problem in the development of infrastructures is that each agent group have developed their own infrastructure in isolation making impossible the connection and the communication among agents using different infrastructures. The development of generic infrastructures which can be reused in the deployment of different systems and permit the rapid development of agent applications has been identified as a fundamental issue for the succed of agent technologies [Ashri and Luck, 2001].

In the last years, several agent platforms have been developed with the aim of solving these problems and facilitating the development of multi-agent applications. In general they offer generic agents with basic functionalities which users should extend and an execution environment which facilitates agent communication at execution time. Most of them follow the standard proposed by the Foundation for Intelligent Physical Agents (FIPA) which defines a set of roles that are mandatory for an agent platform [FIPA, 2001, FIPA, 2002]. Concretely, FIPA proposes that an agent platform must contain at least the following mandatory roles:

- Agent Management System (AMS): controls agents access and use of the platform. The AMS keeps information of all the agents within the platform including their identifiers and transport addresses and it gives a white pages service to the agents connected to the platform. The AMS has control over the lifecycle of the different agents connected to the platform. Agents must register to the AMS when they are connected to the platform in order to obtain a valid identifier within the platform and each agent has a unique identifier within the platform which is normally composed by its name and its home platform address. There is one AMS per platform.

- Directory Facilitator (DF): provides yellow page services to the agent platform. Agents within the platform can register their services to the directory facilitator and can query it in order to know the services offered by other agents.

- Agent Communication channel (ACC): is the default communication method which offers a reliable, orderly and accurate message transport service. The ACC provides agent communication inside and outside the platform.

The principal FIPA-compliant platforms are FIPA-OS [Poslad et al., 2000], JADE [Bellifemine et al., 2001] and ZEUS [Nwana et al., 1999]. Apart from the

services specified by FIPA these platforms offer other facilities such as: tools to develop agents, execution monitoring tools, etc. We believe that the most important contribution of FIPA-compliant platforms to the development of agent applications is that agents on top of any of them can communicate in the same manner independently of which platform each agent is running in. Although they give some support to the use of conversation protocols among agents they fail in managing organisational concepts at execution time.which is necessary in the development of open MAS. For instance, to have control on the roles that agents are playing or on the conversations in which they can engage at each moment.

From our point of view, FIPA-compliant platforms, as they are today, represent a communication layer. On top of them other layers taking into account the organisational structure of multi-agent system must be developed. That is to say, layers which coordinate agents interactions, impose the organizational structure to participating agents and prevent the system from fraudulent behaviours. In our case we have opted for developing a social layer middleware on top of the JADE platform and the participating agents within the institution. JADE is used as a communication layer, making use of its reliable transport service. The social layer is in charge of handling institution concepts at execution time and enforcing the institutional rules to participating agents. For this purpose, the social layer is in charge of controlling the roles that agents are playing, the conversations in which they are participating, and the obligations that each agent has acquired.

We believe that the development of appropriate infrastructures for multi agent systems remains in an earlier stage and a lot of work is needed to permit the rapid development of agent systems. Although FIPA-compliant platforms represent a first step in having standard infrastructures which allow heterogeneous agents to communicate, they fail in coping with the abstract concepts offered by MAS methodologies.

Chapter 3

Electronic Institutions

In this chapter and following the work by [Noriega, 1997, Rodríguez-Aguilar, 2001] we present a formalisation of electronic institutions. We argue that open agent organisations can be modelled as electronic institutions where a vast amount of heterogeneous software and human agents can participate. Human institutions [North, 1990] have been succesfully mediating among humans interactions for centuries and we advocate that electronic institutions may cope with a similar job within agent societies. Electronic institutions define the rules of the game in agent societies in the same way as human institutions do in human societies. That is to say, they define what agents are permitted to do and the consequences of such actions. Furthermore, institutions are in charge of enforcing their rules and punishing those agents that violate them.

What is it needed to define an electronic institution? From our point of view, it is necessary to define a common language, the activity, that is, what can be done within the institution and the consequences that agent' actions within the institution have. As heterogeneous agents, probably developed by different people in different languages and with different architectures, are allowed to participate in the institution, a common language and ontology that permits agents to understand each other must be defined. The activity within the institution is structured in conversations which define the valid interactions that agents can have and the context where the information exchanged must be interpreted. We take a strong dialogical stance in the sense that we understand a multi-agent system as a type of *dialogical system*. In the context of an institution some agent's actions may have consequences that limit or enlarge its acting possibilities. In our case this is captured by the notion of obligation. Then some agent' actions imply the acquirement of some obligations that agents must fulfill later on. In order to abstract from the concrete agents that will participate in the institution, its specification is based on the notion of role, where each role defines a pattern of behaviour within the institution. Then, agents must participate in the institution playing some of its role(s).

Taking into account these requirements our electronic institution model is

based on four elements: dialogic framework, scene, performative structure and norm. The dialogic framework defines the valid illocutions that agents can exchange and which are the participant roles. The institution activity is defined in the performative structure based on the notion of scene. A scene defines a conversation protocol for a group of roles that can be multiply instantiated by different groups of agents playing those roles. Note that all the interactions between participating agents take place within the context of a scene. Thus, a performative structure defines which are the institution scenes (conversations) and how agents, depending on their role and their past actions, can move among them. Finally, norms define the consequences that agents' actions within scenes will have in the future, expressed as obligations.

Before presenting the formalisation of all these elements we want to note that we focus on macro-level (societal) aspects referring to the infrastructure of electronic institutions instead of the micro-level (internal) aspects. We say that we define architectural neutral institution in the sense that no restrictions are imposed on participating agents. That is to say, no restrictions are imposed on the language and architecture used to develop the agents allowing the designer to choose the language and architecture that better fulfills its goals.

Next we present in detail the different elements of our electronic institution model. We start focusing on the dialogic framework, followed by scene, performative structure and finishing with norms.

3.1 Dialogic Framework

In the most general case, each agent immersed in a multi-agent environment is endowed with its own inner language and ontology. In order to allow agents to successfully interact with other agents we must address the fundamental issue of putting their languages and ontologies in relation. For this purpose, we propose that agents share, when communicating, what we call the *dialogic framework* that contains the elements for the construction of the communication language expressions. By sharing a dialogic framework, we enable heterogeneous agents to exchange knowledge with other agents.

The dialogic framework determines the valid illocutions that can be exchanged between the participants. In order to do so, an ontology that fixes what are the possible values for the concepts in a given domain is defined, e.g goods, participants, locations, etc.

Moreover, the dialogic framework defines which are the roles that participating agents may play within the institution. The notion of role is central in the specification of electronic institutions and each role defines a pattern of behaviour within the institution. Roles allow us to abstract from the individuals, the agents, that get involved in an institution's activities. This is specially important in open systems where it is impossible to know in advance which will be the concrete agents that will participate in the institution. Furthermore, the participating agents will change over time, that is, new agents will join the institution and some of the participants will leave. Then, all the actions that

can be done within an institution are associated to roles. Notice that we can think of roles as agent types. More precisely, we define a role as a finite set of actions. Such actions are intended to represent the capabilities of the role. For instance, an agent playing the buyer role is capable of submitting bids and an agent playing the auctioneer role can offer goods at auction. In order to take part in an electronic institution, an agent is obliged to adopt some role(s). Thereafter an agent playing a given role must conform to the pattern of behaviour attached to that particular role. Therefore, all agents adopting a very same role are guaranteed to have the same rights, duties and opportunities.

From the set of roles that can participate in an institution, we differentiate between the internal and the external roles. The internal roles define a set of roles that will be played by staff agents which are like the workers in a human institution. Those agents are in charge of guaranteeing the correct execution of an institution. For instance, an auctioneer is in charge of auctioning goods following the specified protocol and the buyer admiter is in charge of guaranteeing that only buyers satisfying the admission conditions will be allowed to participate with the buyer role within the institution. Then, never an external agent will be allowed to play an internal role; they can only play external roles. Finally, relations over roles can be specified, for instance, roles that can not be played both at the same time.

Definition 3.1.1 We define a *dialogic framework* as a tuple $DF = \langle O, L, I, R_I, R_E, R_S \rangle$ where

- O stands for an ontology (vocabulary);
- L stands for a content language to express the information exchanged between agents;
- I is the set of illocutionary particles;
- R_I is the set of internal roles;
- R_E is the set of external roles;
- R_S is the set of relationships over roles.

Within a dialogic framework the content language allows for the encoding of the knowledge to be exchanged among agents using the vocabulary offered by the ontology. The propositions built with the aid of the content language are embedded into an "outer language", the communication language(CL), which expresses the intentions of the utterance by means of the illocutionary particles. We take this approach in accord to speech act theory [Searle, 1969], which postulates that utterances are not simply propositions that are true or false, but attempts on the part of the speaker that succeed or fail.

We consider that the expressions of the communication language are constructed as formulae of the type $(\iota\,(\alpha_i\,\rho_i)\,(\beta)\,\varphi\,\tau)$ where ι is an *illocutionary particle*, α_i is a term which can be either an agent variable or an agent identifier,

ρ_i is a term which can be either a role variable or a role identifier, β represents the addressee(s) of the message which can be an agent or a group of agents, φ is an expression of the *content language* and τ is a term which can be either a time variable or a time constant. The CL allows to express that an illocution is addressed to an agent, to all the agents playing a role or to all the agents in the scene. If the illocution is addressed to one agent, β is of the form $\alpha_j\, \rho_j$, where α_j is a term which can be either an agent variable or an agent identifier and ρ_j is a term which can be either a role variable or a role identifier. If the illocution is addressed to all the agents of a role, β is of the form ρ_j where ρ_j is a term which can be either a role variable or a role identifier. Finally, if β is equal to the particle "all" it means that the illocution is addressed to all the agents of the scene. We say that a CL expression is an *illocution schema* when some of the terms contain variables. Otherwise, we say that a CL expression is an *illocution*. This distinction will be valuable when specifying scenes in the following section.

Variable identifiers appearing in the illocution schemes can start with either '?' or '!'. As we will see, in subsection 3.2.1, the starting symbol will serve to differentiate when the variable can be bound to a new value or when it must be replaced by its last bound value.

Finally, we would like to stress the importance of the dialogic framework as the component containing the ontologic elements on the basis of which any agent interaction can be specified, as illustrated next, when introducing the notion of scene. Notice, that the dialogic framework determines the set of valid illocutions defining the sets from which the different elements of the CL expressions can take their values. Thus, a dialogic framework must be regarded as a necessary ingredient to specify scenes.

3.2 Scene

As said before, a scene is, in broad terms, a conversation protocol played by a group of agents. More precisely, a scene defines a generic pattern of conversation protocol between roles. Any agent participating in a scene has to play one of its roles. It is generic in the sense that it can be repeatedly played by different groups of agents. In the same sense that the same theater scene can be performed by different actors incarnating the same scene characters.

A scene protocol is specified by a finite state directed graph where the nodes represent the different states of the conversation and the directed arcs connecting the nodes are labelled with the actions that make the scene state evolve. These are: illocution schemes and timeouts. The graph has a single initial state (non-reachable once left) and a set of final states representing the different endings of the conversation. There is no arc connecting a final state to some other state.

Apart from defining the valid sequences of illocutions that agents can exchange, a scene keeps the conversation context. Context is a fundamental aspect that humans use in order to interpret the information they receive. The same message in different context may certainly have a different meaning. Thus, a scene keeps what has been said by who and to whom, and allows to specify how

past interactions may affect the future evolution of the conversation. The contextual information may restrict the valid messages in a certain moment (state) of the conversation. That is to say, the same message in the same state may not be valid because the context information has changed. For instance, imagine a scene auctioning goods following the English auction protocol. As bids are submitted by buyers the valid bids for them are reduced to bids greater than the last one. That is to say, each submitted bid reduces the valid illocutions that buyers can utter, although the scene may continue in the same state.

Because we aim at modeling multi-agent conversations whose set of participants may dynamically vary, scenes will allow that agents either join in or leave at some particular moments (states) during an ongoing conversation depending on their role. For this purpose, we differentiate, for each role, the sets of access and exit states. Normally the correct evolution of a conversation protocol requires a certain number of agents for each role involved in it. Thus, a minimum and maximum number of agents per role is defined and the number of agents playing each role has to be always between them. This restriction must be taken into account in order to allow agents to join or leave the conversation. Obviously, the final states have to be an exit state for each role, in order to allow all the agents to leave when the scene is finished. On the other hand, the initial state has to be an access state for the roles whose minimum is greater than zero, in order to start the scene.

The information exchanged between agents is expressed in the form of illocution schemes from the scene dialogic framework. In order for the protocol to be generic some details have to be abstracted. This means that state transitions cannot be labelled by grounded illocutions. Instead what shall be used are illocutions schemes where, at least, the terms referring to agents and time must be variables while the other terms can be variables or constants[1]. As mentioned above, we want the conversation protocols to be generic, that is, independent of concrete agents and time instants.

The other element that can label an arc is a timeout. Timeouts allow to provoke transitions after a given number of time units have passed since the state was reached. This is specially important for robustness (to evolve from states where agents dying and hence not talking any more or where agents trying to foot-drag the other agents by remaining silent could block the scene execution). One important point is that if there is an arc from one state to itself, this transition does not affect the timeout. The timeout countdown starts when the state is reached and stops only if there is a transition to a different state of the scene. Using timeouts, you can for instance make the scene state evolve from a state where bids are accepted to a state where the bidding time is over. A timeout is a numeric expression composed by numeric constants and bound numeric variables (that is, starting with '!'). This permits that timeouts may change during the evolution of the scene as a consequence of agents' interaction. For instance, the bidding time can be different at each bidding round.

[1] The variable refering to time will be omitted on this document as their usage is straightforward

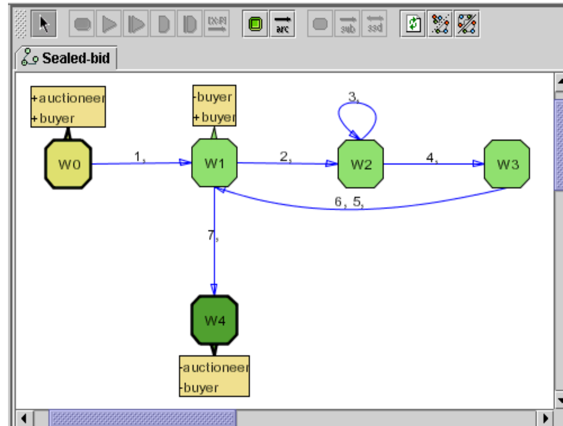


Figure 3.1: Specification of the sealed-bid auction protocol

As an example of a scene Figure 3.1 presents the specification of the conversation protocol of an auction scene realising a sealed-bid protocol. Next you can see the value of the labels associated to the different arcs:

1 $(inform\ (?x\ auctioneer)\ (buyer)\ open_auction(?r))$
2 $(inform\ (!x\ auctioneer)\ (buyer)\ start_round(?good_id, ?bidding_time,$
$?reserve_price))$
3 $(commit\ (?y\ buyer)\ (!x\ auctioneer)\ bid(!good_id, ?offer))$
4 $!bidding_time$
5 $(inform\ (!x\ auctioneer)\ (buyer)\ sold(!good_id, ?price, ?winner))$
6 $(inform\ (!x\ auctioneer)\ (buyer)\ withdrawn(!good_id))$
7 $(inform\ (!x\ auctioneer)\ (buyer)\ end_auction(!r))$

In this scene the participating agents can play the auctioneer and buyer roles. Concretely, the scene specification requires the participation of exactly one agent playing the auctioneer role and imposes no restriction on the number of buyers. The graph depicts the states of the scene, along with the edges representing the legal transitions between scene states which are labelled either with illocution schemes of the communication language or with timeouts. The information content of such schemes is expressed in prolog. Notice that apart from the initial and final states, the $w1$ state is labelled as an access and exit state for buyers —meaning that between rounds buyers can leave and new buyers might be admitted into the scene.

In a sealed-bid protocol buyers have a specified period of time to submit their bids and after that period the auctioneer announces the winner who will be the buyer who submitted the highest bid. A round starts at $w1$ with a broadcast message from the auctioneer to all the buyers. This illocution brings the information of the good identifier expressed as a value for the $?good_id$ variable, the time that buyers have to submit their bids expressed on the $?bidding_time$

variable and the reserve price for this good expressed on the $?reserve_price$ variable. The scene will remain in $w2$ until the timeout of label 4 expires and the scene evolves to $w3$. This timeout corresponds to the bidding time announced by the auctioneer in the previous illocution. During this time buyers can submit their bids uttering illocutions matching the illocution schema of label 3. At state $w3$ the auctioneer will announce the result of the round, labels 5 or 6, finishing the round and evolving back to $w1$. Concretely, label 6 corresponds to the case where no buyer has submitted a bid in this round and the auctioneer declares the lot withdrawn, and label 5 corresponds to the case that buyers have submitted bids and then the auctioneer announces the winner and the price it has to pay for the good. We want to remark that although the arc connecting $w3$ and $w1$ has associated two labels 5 and 6 this is a disjunction and never both can be uttered at the same round. In other words, the utterance of an illocution matching the illocution scheme in label 5 or the illocution scheme in label 6 will make the scene evolve to $w1$. At $w1$ the auctioneer can start another round or can declare the finalisation of the scene making the scene evolve to its final state $w4$.

Next we focus on how the variables appearing in the specification of a scene are handled when the scene is played.

3.2.1 Variables

Recall that the arcs connecting the states of a scene are labelled with illocution schemes containing variables. During a scene conversation, the variables in illocution schemes are bound to the values of the uttered illocutions. In order to verify the correctness of subsequent illocutions we need to keep track of these bindings because they represent the context of the interaction. Then, they may restrict the values of the arguments of subsequent illocutions and the paths that the scene execution can follow. These bindings change dynamically, that is, the same variable may be bound to different values at different stages of the conversation within a scene. Notice that the type of a variable in the context of a scene must be the same for all occurrences.

For instance, in a protocol for iterated negotiation between two agents we might be interested to refer to the last proposal of each agent. To do this we use the same variable for all transitions in the protocol referring to an agent's proposal. We need to distinguish when the variable occurrence is to be bound (to the actual illocution uttered by an agent) and when the actual value appearing in the agent's uttered illocution has to match the variable's last bound value. For instance, when an agent makes a new proposal a new variable binding can be created and when an agent accepts the last proposal of the other agent we want the value of the variable in the accept illocution to be actually the value of the last proposal of the other agent. This is, the last binding of the variable representing the other agent's proposal. In order to model this we use two different prefixing symbols of the variable identifier. When the variable is preceded by a '?' it is a binding occurrence and it can be bound to any value of its type, when the variable is preceded by a '!' it is an application occurrence (we refer to its value).

An application occurrence must be preceded by at least one binding occurrence for the protocol to be correct; at the beginning of the scene all variables are unbound. Variable's scope is the scene in which they appear.

The utterance of an actual illocution during the conversation will be matched against the illocution schemes outgoing the current state. This matching will generate a substitution $\sigma_{w_i w_j}$ for those variables in the scheme prefixed by '?', where w_i and w_j stands for the source and target state of the transition. For instance, the submission of a bid in the sealed-bid scene of figure 3.1 by agent *John* playing the buyer role by means of the illocution (*commit* (*John buyer*) (*James auctioneer*) *bid*($g1, 25$)) matching the illocution schema (*commit* (?*y buyer*) (!*x auctioneer*) *bid*(!*good_id*, ?*offer*) of label 3 between the states $w2$ and $w2$, will generate the substitution $\sigma_{w2\,w2} = [?y/John, ?offer/25]$. Notice that for the illocution to be correct the last binding for variables x and *good_id* must be *James* and $g1$ respectively. Thus, we can define Σ as the sequence of all the substitutions done during a conversation, i.e. a sequence of $\sigma_{w_i w_j}$ each one corresponding to an uttered illocution.

As we have said, a variable prefixed by '!' denotes the last bound value of the variable. This is easy to compute by searching backwards in Σ for the first substitution in which the variable appears. Moreover, as we have in Σ all the bindings established during the conversation we can obtain a subset or all the past bindings for a concrete variable. This will be valuable, as we will see in next section, when specifying constraints.

3.2.2 Constraints

In this subsection we will explain the use of constraints to model how past illocutions affect a scene's future evolution. In practical terms, constraints will be used to restrict the set of values to create new bindings of the variables in the illocution schemes, as well as the paths that a scene conversation can follow. Sometimes constraints can completely fix a concrete value and sometimes they just restrict the set of possible values. For instance, in a sealed-bid scene the value of any bid has to be higher or equal than the reserve price, or when the auctioneer utters the illocution declaring the winner, the value of the variable representing the winner has to be the identifier of the agent who has submitted the highest bid. Constraints can also restrict or determine the paths that the conversation can follow. For instance, an auctioneer is only allowed to declare the lot withdrawn if no bids have been submitted in the current round.

As we said, we store in Σ the sequence of substitutions produced during the conversation. Prefix '!' allows to obtain the last binding for a concrete variable. But in some occasions we need to access several past bindings of a variable. For instance, to check that the auctioneer announces the correct winner and price for a bidding round. To do so we need to recover the bindings for each submitted bid in the last round and look which is the highest bid and the buyer who has submitted this bid. The expression $!_{w_i w_j} x$ will return *all* the bindings for variable x in the substitutions generated by the illocutions that lead from state w_i to w_j including loops over w_i, w_j and any intermediary state. To compute

this we simply search backwards in Σ looking for the first appearance of w_j and returning all the bindings for x from there and until the first transition to w_i from another state. In general we may be interested in obtaining the bindings of the last i times that the conversation evolved from w_i to w_j. We note that with a super index: '$!^i_{w_i w_j}$'. Also, we permit to write conditions that restrict the set of returned bindings. For instance, this allows to select from the bindings of the variable denoting the buyer in bid illocutions, the one corresponding to the highest bid.

We here summarise the meaning of the different prefixes of a variable identifier within constraints:

- $?x$: stands for the value to be bound to variable x as a consequence of the utterance of an illocution matching the illocution scheme labelling the arc to which the constraint is associated. The illocution scheme labelling the arc to which the constraint is associated must contain an occurrence $?x$.
- $!x$: stands for the last binding of variable x.
- $!^i_{w_i w_j} x$: stands for the mulitset[2] of all the bindings of variable x in the i last subdialogues between w_i and w_j. $!^1_{w_i w_j} x$ is noted as $!_{w_i w_j} x$ for simplicity.
- $!^*_{w_i w_j} x$: stands for the multiset of the bindings of variable x in all subdialogues between w_i and w_j in Σ.
- $!^i_{w_i w_j} x$ (*cond*): stands for the multiset of all the bindings of variable x in the i last sub-dialogues between w_i and w_j such that the substitution σ where the binding appears satisfies the condition *cond*.

Let's now concentrate on how to specify constraints. Constraints have this form: *op* $expr_i$ $expr_j$ where expressions are formed as Lisp expressions over the scene variables. The expressions must be of the following basic types: string, numeric and boolean, or a multiset of any of these types, where the operations are:

- $=, \neq, <, <=, >=, >$: $numeric \times numeric \rightarrow boolean$
- $=, \neq$: $string \times string \rightarrow boolean$
- $=, \neq, \vee$: $boolean \times boolean \rightarrow boolean$
- $\in, \notin$: $\alpha \times \alpha\, multiset \rightarrow boolean$, where α is any basic type.
- $\subset, \subseteq$: $\alpha\, multiset \times \alpha\, multiset \rightarrow boolean$, where α is any basic type.

There is a special type of sets that can be used on the constraints. This is the sets of agents playing a role which are expressed by the role identifier. For instance, this allows to specify that the auctioneer can not start a round if there

[2] A multiset is a set where elements can be repeated

are less than two buyers in the auction scene or that all the agents of a role have uttered a concrete illocution.

Each arc labelled with an illocution scheme may have some constraints associated to it that must be satisfied for the transition to happen. Next we present the constraints associated to the labels of the arcs of the sealed-bid scene presented above specified by Figure 3.1:

- $(> |buyer|\ 2)$: this constraint checks that there are at least two agents playing the buyer role before starting a new round. This constraint is associated to label 2.
- $(>=\ ?\mathit{offer}\ !reserve_price)$: forces buyers to submit bids higher or equal than the reserve price uttered by the auctioneer. This constraint is associated to label 3.
- $(notin\ ?y\ !_{w_1 w_2} y)$: this constraint checks that each buyer does not submit more than one bid in a round. The variable $?y$ refers to the buyer which tries to submit a bid while the second expression returns all the bindings for the variable y in this round, i.e. all the buyers who submitted a bid. This constraint is also associated to label 3.
- $(=\ ?price\ (max\ !_{w_2 w_3} \mathit{offer})$: it controls that the price associated to the winner corresponds to the greatest bid submitted by buyers in this round. This constraint is associated to label 5.
- $(in\ ?winner\ (!_{w_2 w_3} y\ (?\mathit{offer} = max(!_{w_2 w_3} \mathit{offer}))))$: this constraint checks that the buyer declared as the winner of the round represented by the variable $?winner$ corresponds to a buyer who has submitted the greatest bid, that is the substitution is of the type $[?y/a, ?\mathit{offer}/v]$ and $v = max(!_{w_2 w_3} \mathit{offer})$. This constraint is also associated to label 5.
- $(> |!_{w_2 w_3} y|\ 0)$: this constraint checks that at least one buyer has submitted a bid in the last round. It must be satisfied in order to declare one of the buyers as a winner of the round and it is associated to label 5.
- $(=\ |!_{w_2 w_3} y|\ 0)$: this constraint checks that no buyer has submitted a bid in the last round. It must be satisfied in order to declare the lot withdrawn. This constraint is associated to label 6.

In conclusion, for any illocution uttered by an agent to be valid it has to match an illocution scheme of an outgoing arc of the current state, it has to respect the bindings of bound variables, and it has to satisfy the constraints. The matching process against the labels of the outgoing arcs can be viewed as a sintactic verification of the illocution uttered by the agent, while the verification of the bindings with respect to the bound values and the satisfaction of constraints can be seen as the semantic verification of the illocution. Next definition summarises the components introduced so far.

Definition 3.2.1 Formally, a scene is a tuple:

$$s = \langle R, DF, W, w_0, W_f, (WA_r)_{r \in R}, (WE_r)_{r \in R}, \Theta, \lambda, min, Max \rangle$$

where

- R is the set of roles of the scene;
- DF is a dialogic framework defined as in 3.1.1;
- W is a finite, non-empty set of scene states;
- $w_0 \in W$ is the initial state;
- $W_f \subseteq W$ is the non-empty set of final states;
- $(WA_r)_{r \in R} \subseteq W$ is a family of non-empty sets such that WA_r stands for the set of access states for the role $r \in R$;
- $(WE_r)_{r \in R} \subseteq W$ is a family of non-empty sets such that WE_r stands for the set of exit states for the role $r \in R$;
- $\Theta \subseteq W \times W$ is a set of directed edges;
- $\lambda : \Theta \longrightarrow L$ is a labelling function, where L can be a timeout, an illocution scheme or an illocutions scheme and a list of constraints;
- $min, Max : R \longrightarrow I\!N$ $min(r)$ and $Max(r)$ return respectively the minimum and maximum number of agents that must and can play the role $r \in R$;

3.3 Performative Structure

While a scene models a particular multi-agent dialogic activity, more complex activities can be specified by establishing relationships among scenes. This issue arises when conversations are embedded in a broader context, such as, for instance, organisations and institutions. If this is the case, it does make sense to capture the relationships among scenes. For these purpose the performative structure defines which are the conversations (scenes) of the electronic institution and the *role flow policy* among them. That is to say, *how* the agents depending on their role can move among the different scenes and *when* new conversations will be started, taking into account the relationships among the different scenes. In order to capture these relationships we use a special type of scenes called *transitions*. The type of transition allows to express agents' synchronisation, chooses points where agents can decide which path to follow or parallelisation points where agents are sent to more than one scene. Transitions can be seen as a kind of routers in the context of a performative structure.

In general, the activity represented by a performative structure can be depicted as a collection of multiple, concurrent scenes. Agents navigate from scene

to scene constrained by the rules defining the relationships among scenes. Moreover, the very same agent can be possibly participating in multiple scenes at the same time. Although this is usually impossible for humans in actual institutions it is very reasonable, and easy to implement, capability of autonomous agents. Hence, it is our purpose to propose a formal specification of performative structures expressive enough to facilitate the specification of such rules.

The way agents move from scene to scene depends on the type of relationship holding among the source and target scenes. As mentioned above, sometimes we might be interested in forcing agents to synchronise before jumping into either new or existing scene executions, offer choice points so that an agent can decide which path to follow or paralelisation points where agents are sent to more than one scene. Summarising, in order to capture the type of relationships listed above we consider that any performative structure contains a special element that we call *transition*, devoted to mediate different types of connections among scenes. Each scene may be connected to multiple transitions, and in turn each transition may be connected to multiple scenes. In both cases, the connection between a scene and a transition is made by means of a directed arc. Then we can refer to the source and target of each arc. And given either a scene or a transition, we shall distinguish between its incoming and outgoing arcs. Notice that there is no direct connection between two scenes, or, in other words, all connections between scenes are mediated by transitions. Also, we do not allow the connection of transitions.

We define a set of different types of transitions and arcs whose semantics will highly constrain the mobility of agents among the scene instances (the ongoing activities) of a performative structure. The differences between the diverse types of transitions that we consider are based on how they allow to progress the agents that they receive towards other scenes. We have defined two types of transitions:

- **And**: They establish synchronisation and parallelism points since agents are forced to synchronise at their input to subsequently follow the outgoing arcs in parallel.
- **Or**: They behave in an asynchronous way at the input (agents are not required to wait for others in order to progress through), and as choice points at the output (agents are permitted to select which outgoing arc, which path, to follow when leaving).

According to this classification, we define $\mathcal{T} = \{And, Or\}$ as the set of transition types.

The arcs connecting transitions to scenes play a fundamental role. Notice that as there might be multiple (or perhaps none) scene executions of a target scene, it should be specified whether the agents following the arcs are allowed to start a new scene execution, whether they can choose a single or a subset of scenes to incorporate into, or whether they must enter all the available scene executions. Thus, there are also different types of paths, arcs, for reaching scenes after traversing transitions. We define $\mathcal{E} = \{1, some, all, new\}$ as the set of arc types. Following a *1-arc* constrains agents to enter a single scene instance of

the target scene, whereas a *some-arc* is less restrictive and allows the agents to choose a subset of scene instances to enter, and an *all arc* forces the agents to enter all the scene instances to which the paths lead. Finally, a *new arc* fires the creation of a new scene instance of the target scene.

The label of each arc of the graph determines which agents depending on their role can progress through the arc. This is expressed as conjunctions and disjunctions of pairs of an agent variable and a role identifier. The role identifier determines which agents will be allowed to follow the arc depending on their role while the agent variables are used to differentiate among agents playing the same role. For instance, a label $(x\,R_1) \wedge (y\,R_2)$ means that this arc can be followed by pairs of agents where one of them is playing the role R_1 and the other is playing the role R_2. On the other hand, a label as $(x\,R_1) \vee (y\,R_2)$ means that any agent playing one of the roles R_1 or R_2 can progress through the arc alone. Concretely, each arc is labeled with a disjunctive normal form of pairs of agent variable and role identifier. In general if we have a label $L_1 \vee \ldots \vee L_n$ each L_i determines a set of agents playing the defined roles that can progress through the arc. The scope of the agent variables is the incoming and outgoing arcs of the transition. That is to say, if an agent reaches a transition following an arc labelled with $(x\,R_1)$ it can only leave the transition following those arcs which contain the variable x in their label. On the contrary, there is no relation between the agent variables labelling the incoming and the outgoing arcs of a scene. An important point is that, when the arc is connecting a scene to a transition, a conjunction means that agents have to leave the source scene together, meanwhile, when, the conjunction labels an arc from transition to scene it means that agents must incorporate to the same scene execution(s).

Agents will be moving from a scene instance (execution) to another by traversing the transition connecting the scenes and following the arcs that connect transitions and scenes. Transitions must be regarded as a kind of routers within the performative structure. Therefore, instead of modelling some activity, they are intended to route agents towards their destinations in different ways, depending on the type of the transition. Thus, when in a transition agents can ask about its possible destinations and request which path they want to follow and/or which scene instance(s) they want to join. The possibilities that an agent has, depend on the type of the transition and on the type and labels of the transition outgoing arcs. For instance, in an *Or* transition if there is more than one arc that an agent can follow it must choose only one to follow, while in an *And* transition it has no choice, and the agent will follow all of them. On the other hand, an outgoing arc of type *new* means that agent(s) will be incorporated to a newly created scene execution, while an outgoing arc of type *one* means that agents must select one of the current executions of the target scene to incorporate into. Thus, transitions constitute an intermediate state for agents that move from scene to scene and the interaction whithin them is concerned to agents' destinations within the performative structure.

From a structural point of view, performative structures' specifications must be regarded as networks of scenes mediated by transitions. At execution time, a

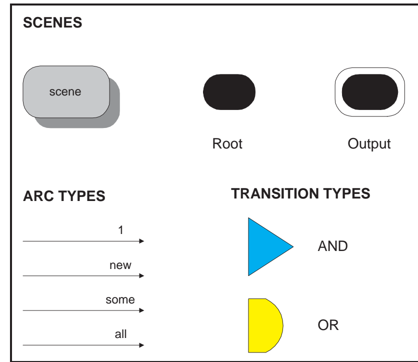


Figure 3.2: Graphical Elements of a Performative Structure

performative structure becomes populated by agents that make it evolve whenever these comply with the rules encoded in the specification. Concretely, an agent participating in the execution of a performative structure devotes his time to jointly start new scene executions, to enter active scenes where the agent interacts with other agents, to leave active scenes to possibly enter other scenes, and finally to abandon the performative structure.

Figure 3.2 depicts the graphical representations that we will employ to represent the performative structure's components introduced so far. From the point of view of the modeller, such graphical components are the pieces that serve to construct graphical specifications of performative structures.

Before presenting the definition of performative structure, there is a last element to be considered. Notice that although two scenes may be connected by a transition, the eventual migration of agents from a source scene instance to a target scene instance not only depends on the role of the agents but also on the results achieved by agents in previous scenes. Thus, for instance, in an auction house, although a registration scene is connected to an auction scene, the access of a buying agent to the execution of an auction scene is forbidden if it has not successfully completed the registration process when going through a registration scene. This fact motivates the introduction of constraints over the arcs connecting scenes and transitions. The basic elements to define constraints are illocution schemes and obligation predictaes. Then, a negation of them, a disjunction or a conjunction of some of them can appear in a constraint definition. We will require that agents satisfy the constraints, conditions, over the arc solicited to be followed when attempting to leave a scene. Therefore, conditions must also appear in our formal definition of a performative structure.

We bundle all the elements introduced above to provide a formal definition of a performative structure specification:

Definition 3.3.1 A performative structure is a tuple

$$PS = \langle S, T, s_0, s_\Omega, E, f_L, f_T, f_E^O, C, \mu \rangle$$

where

- S is a finite, non-empty set of scenes defined as in 3.2.1;
- T is a finite and non-empty set of transitions;
- $s_0 \in S$ is the *initial* scene;
- $s_\Omega \in S$ is the *final* scene;
- $E = E^I \bigcup E^O$ is a set of arc identifiers where $E^I \subseteq S \times T$ is a set of edges from scenes to transitions and $E^O \subseteq T \times S$ is a set of edges from transitions to scenes;
- $f_L : E \longrightarrow FND_{2^{V_A \times R}}$ maps each arc to a disjunctive normal form of pairs of agent variable and role identifier representing the arc label;
- $f_T : T \longrightarrow \mathcal{T}$ maps each transition to its type;
- $f_E^O : E^O \longrightarrow \mathcal{E}$ maps each arc to its type;
- $C : E^I \longrightarrow CONS$ maps each arc to a expression representing the arc's constraints[3].
- $\mu : S \longrightarrow \{0, 1\}$ sets if a scene can be multiply instantiated at execution time;

Notice that we demand any performative structure to contain an initial and a final scene. The final scene does not model any activity, and so it must be regarded as the exit point of the performative structure. As to the initial scene, it must be regarded as the starting point of any agent accessing the performative structure. Departing from the initial scene, agents will make for other scenes within the performative structure.

3.4 Norms

As described so far, the performative structure defines what participating agents are *permited* to do within the institution depending on their role. The performative structure constrains the behaviour of participating agents at two levels:

- *intra-scene:* Scene protocols dictate for each agent role within a scene what can be said, by whom, to whom, and when.
- *inter-scene:* The connections between the scenes of a performative structure define the possible paths that agents may follow depending on their roles. Furthermore, the constraints over output arcs impose additional restrictions on agents attempting to reach a target scene.

[3]Reader is referred to 5.1.4 for a sintactical definition of the arc constraints expressions

But some agent's actions *within* scenes may have consequences that either limit or expand its possible subsequent actions, outside the scope of the scene. The consequences we have identified can take two different forms. Some actions create commitments for future actions, which may be interpreted as obligations. Other actions may affect the paths an agent may take through the performative structure because it may change which constraints are satisfied. For instance, a trading agent winning a bidding round within an auction house is obliged to pay later on for the acquired good before leaving the institution. In order to capture these consequences, we use a special type of rules called *norms*.

Norms must define the actions that will provoke its activation, the obligations that agents will have and the actions that agents must carry out in order to fulfil the obligations. As we are specifying dialogical institutions, agents actions are expressed as a pair of illocution scheme and scene where it is uttered. We need both components because the same illocution could appear in more than one scene. The scene gives the context in which the illocution must be interpreted and of course, this affects the consequences that the utterance of the illocution has. That is to say, the same illocution may have different consequences in different scenes because it is uttered in a different context. As we have said some of the terms of an illocution scheme are variables. The activation of a norm may depend on the values of these variables in the uttered illocutions. Then, some conditions on the value of these variables can be imposed. These conditions are specified as boolean expressions over illocution scheme variables and a norm will not be activated if they are not satisfied. The scope of the variables is the complete norm.

In order to represent the deontic notion of obligation we set out the predicate *obl* as follows:

$$obl(x, \psi, s) = \text{agent}\, x \,\text{is obliged to do}\, \psi \,\text{in scene}\, s.$$

where ψ is taken to be an illocution scheme. We denote the set of obligations by Obl and any concrete obligation by $obl_i \in Obl$. Norms follow the schema:

$$(s_1, \gamma_1) \wedge \ldots \wedge (s_m, \gamma_m) \wedge e_1 \wedge \ldots \wedge e_n \wedge$$
$$\wedge \neg((s_{m+1}, \gamma_{m+1}) \wedge \ldots \wedge (s_{m+n}, \gamma_{m+n})) \rightarrow obl_1 \wedge \ldots \wedge obl_p$$

where $(s_1, \gamma_1), \ldots, (s_{m+n}, \gamma_{m+n})$ are pairs of scenes and illocution schemes, $e_1, \ldots e_n$ are boolean expressions over illocution schemes' variables, $\neg$ is a defeasible negation, and $obl_1, \ldots, obl_p$ are obligations. The meaning of these rules is that if the illocutions $(s_1, \gamma_1), \ldots, (s_m, \gamma_m)$ have been uttered, the expressions $e_1, \ldots, e_n$ are satisfied and the illocutions $(s_{m+1}, \gamma_{m+1}), \ldots, (s_{m+n}, \gamma_{m+n})$ have *not* been uttered, the obligations $obl_1, \ldots, obl_p$ hold. Therefore, the rules have two components, the first one is the causing of the obligations to be activated (for instance winning an auction round by saying 'mine' in a downwards bidding protocol, generating the obligation to pay) and the second is the part that removes the obligations (for instance, paying the amount of money due for the round which was won).

Clearly, an external agent might not fulfil its obligations. As agents are autonomous and the institution accepts agents developed by other people, those agents cannot be forced to utter particular illocutions. From this follows that institutions cannot force agents to fulfil their obligations. However, the institution knows the obligations that each agent has acquired and can thus detect when an agent does not fulfil its obligations and hence violates the norms. Moreover, the institution can restrict the actions that an agent can carry out while it has not fulfilled some or all of its obligations.

3.5 Electronic Institution

Taking into account the three main concepts, which we pointed out, as necessary for defining an institution: language, activity and consequences, an institution is defined by a dialogic framework, a performative structure and a list of norms.

Definition 3.5.1 An electronic institution is defined as a tuple

$$EI = \langle \mathcal{DF}, \mathcal{PS}, N \rangle$$

where

- $\mathcal{DF}$ stands for a dialogic framework;
- PS stands for a performative structure;
- N stands for a set of norms.

3.6 Conclusions

In this chapter we have presented how open multi-agent organizations can be formalised as electronic insitution. Although organisational design is widely admitted as a fundamental issue in multi-agent systems, social concepts have been introduced in a rather informal way [Ferber and Gutknetch, 1998]. Hence the need for formally incorporating organisational terms and concepts into multi-agent systems. Due to the complexity of this type of systems, we have adopted a formal approach and we defend that the development of electronic institutions must be preceded by a precise specification that fully characterise the institution's rules. In general, the presence of an underlying formal method underpins the use of structured design techniques and formal analysis, facilitating development, composition and reuse. For this purpose, we have presented a formalisation of electronic institutions which continues the work of [Noriega, 1997, Rodríguez-Aguilar, 2001] and where we have refined and extended the definition of some of the components. We believe that our principal contribution to institutions' formalisation has been done at scene level by the introduction of time-outs and constraints. Since time-outs permits to make the

scene evolve when no agent is talking, constraints permit to capture the consequences of the previous interaction whithin a scene in its furure evolution. The analysis required for the complete formalisation of the system allows the modeller to gain a dramatically improved understanding of the modelled institution before developing it. Also, it permits to detect the critical points of the system and detect errors at an early stage.

As we have said we focus on macro-level aspects of agents. That is, which is the language that agents will use to communicate, the valid interactions they may have and the consequences of these interactions. We structure all agent interactions in conversations which define the valid interactions and give the context where exchanged messages must be interpreted. Then, the formalisation determines at every moment what can be done by each of the participants. That is to say, what can be said, by who and to whom within each conversation and the valid movements that agents can do among the different conversations.

In the light of the complexity of the whole process, it is apparent the need of tools that assist the institution designer through the specification, validation, and generation of infrastructures for the specified institutions. The formalisation presented here will be the basis for the development of software tools that will help institution designers. Thus, in chapter 5 we will focuss on the tool for the specification and verification of electronic institutions while in chapter 6 we will focuss on how infrstarucutres for the specified insitution are generated from institution specifications.

Chapter 4

Formalising Institutions in Process Algebras

Research on the formalisation of concurrent and distributed systems has quite long history starting with Milner's Calculus of Communicating Systems (CCS) [Milner, 1980] and Hoare's Communicating Sequential Processes (CSP) [Hoare, 1985]. Their goal is to define a formal framework where concurrent and distributed systems can be specified and then, analysed and verified. These languages and their successors focus on which processes compose a system and how they communicate.

In the work reported in this chapter, we have used an extension of CCS to formalise multi agent systems. Concretely we have used the π-calculus to define a distributed bidding resolution mechanism within an auction room [1] The π-calculus [Milner, 1999] extends CCS by allowing the definition of polyadic channels and to pass channels over channels.

We advocate that each participating agent within an institution is connected to a special type of agent, the so-called *governor* which mediates its communication with the rest of the agents. This permits agents to abstract from communication problems and concentrate on how to take their decisions. Furthermore governors control that agents behave according to the institution rules. In order to distribute the bidding resolution mechanism for the winner determination of auction protocols we make use of the governors. For this purpose, the governors of the buyers taking part in the auction are extended to be able to resolve the winner determination process of auction rounds in a distributed way. So, the governors of the buyers taking part in the auction coordinate to decide which of them is the winner of each auction round. It is quite natural for electronic institutions to follow the structure of their physical counterparts. However, this is not always appropriate or desirable in a virtual setting. We report on the

[1] The work presented in this chapter has been done in collaboration with Julian Padget and has been extracted from [Esteva and Padget, 2000]. We want to thank him for authorising to use material from the paper.

prototyping of an alternative architecture for electronic auctions based around the concept of a *governor* and building on the considerable body of work in the distributed algorithms literature to plot a path toward *resilient* trading frameworks. In particular, we have adapted the classical Leader Election algorithm for resolving bids in a generic auction scheme as well as identifying the factors which differentiate the physical auction protocols in such a way that new auction protocols can be plugged into the scheme by the specification of the relevant (sub-)processes. We have used the π-calculus to specify both the generic scheme and the specific protocols of first-price, second-price, Dutch and English. The bid resolution process has been prototyped in Pict [Pierce and Turner, 1997].

4.1 Distributed winner determination process

Electronic commerce has become more and more important with the growth of the Internet. In particular, auctioning has become one of the most popular mechanisms of electronic trading, as we can see from the proliferation of on-line auctions on the Internet. Multi-agent systems appear to offer a convenient mechanism for automated trading, due mainly to the simplicity of their conventions for interaction when multi-party negotiations are involved. AI researchers have been interested in two areas: auction marketplaces and trading agents' strategies and heuristics [Garcia et al., 1998, Varian, 1995, Ygge and Akkermans, 1997]. Apart from web-based trading, auctions are the most prevalent co-ordination mechanism for agent-mediated resource allocation problems such as energy management [Ygge and Akkermans, 1997, Ygge and Akkermans, 1996], climate control[Huberman and Clearwater, 1995], flow problems[Huberman and Clearwater, 1995], computing resources [Gagliano et al., 1995], public monopolies [Bushnell and Oren, 1993] and many others [Clearwater, 1995].

From the point of view of multi-agent interactions in auction-based trading, the situation is deceptively simple. Trading within an auction house demands that buyers merely decide an appropriate price to bid, and that sellers essentially only have to choose the time to submit their goods.

The work related here is a continuation of the FishMarket (FM) project[Rodríguez-Aguilar et al., 1997, Rodríguez-Aguilar et al., 2000] and on the work on formalising the auction house in π-calculus presented in [Padget and Bradford, 1998]. In this work we focus on the auction room, and especially the process used for the resolution of bids. In all the versions of the FM to date, this process has been carried in a centralized manner. While that corresponds to the physical reality of auctions, it is not necessarily an appropriate model in a computing context due to two problems that are not common in physical situations (*pace* telephone bidders!): breakdown of processes or communications — so-called stopping failures — and intermittently faulty processes or communications, leading to unreliable messages — so-called Byzantine failures [Lamport et al., 1982]. As a first step for addressing these problems we here present a distributed solution to which other techniques may

later be added to handle resilience issues, which effectively does away with the auctioneer, thus removing a central single point of failure. In some sense the governors — the market interfaces for the buyers —, as we shall see later, are really replications of the auctioneer, in common with the architecture proposed in [Franklin and Reiter, 1996].

In the current FM all the bids are submitted to an agent called the auctioneer who controls the whole auctioning process and determines the result of the round. From this point of view, the resolution of the bidding protocol is centralized. What we do in this new version is to distribute this process among the buyers' market interface agents (called governors) — see Figure 4.1. Thus, the auctioneer sends the buyers (via the governors) the information about the lot and then waits until one buyer governor sends it the result of the round. During the intervening period the buyer governors resolve the bids using a distributed protocol. Thus, the work of the auctioneer is reduced to:

- controlling which buyers participate in the auction,
- starting the rounds by sending the information on the lots
- waiting for the result of the round

We have two motivations: to have another way of applying auction protocols and to have a way to avoid the auctioneer becoming a bottleneck. With this new mechanism, the load of messages is distributed between all the governors. The idea is that this algorithm can be applied to an existing auction house, providing an alternative way to deliver different auction protocols.

The basis of the distributed approach is the Leader Election algorithm [Lynch, 1996]. It can be understood simply as an algorithm for choosing one processor in a network of many and the version we have adapted here derives from that used in token ring networks, where it is used to regenerate the token.

In the next section 4.1.1 we explain the new organization of the auction room in order to apply the algorithm. In section 4.1.2 we give a brief summary of π-calculus. In section 4.1.3 we outline how to apply the Leader Election algorithm in auction protocols. Finally, in sections 4.1.4 to 4.1.8 we explain how to use this algorithm for a range of auction protocols and we give their specification in π-calculus too.

4.1.1 Organization of the auction room

There are two kind of agents taking part in the auction room scene: The auctioneer is an institutional agent who controls the correct running of the auctions. It controls when buyers enter and leave, starts the rounds and receives the result of them. Finally, it declares the end of the auctions when it has no further lots to auction.

We have added another kind of agent, the so-called *governor*, an autonomous software agent which mediates interactions between a buyer and the agent society wherein it is situated. A governor is connected to a buyer and abstracts it from

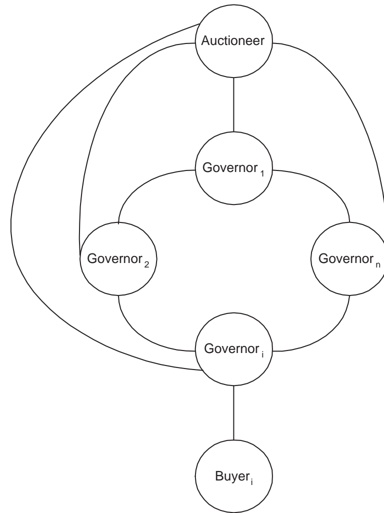


Figure 4.1: Organization of the agents in the auction room

communication problems. Thus, each buyer only has to communicate with its governor so it can focus on its strategies for bidding. The use of a governor also has advantages for the institutions, because the governor forces the buyer to follow the auction room protocol. For example, the governor will prevent the buyer from making a bid between rounds. The governors also implement the resolution of the bidding protocol. Previously, this task was done by the auctioneer, whereas now it is distributed among the governors, which are linked in a ring in order to apply the particular version of Leader Election we have chosen [Lynch, 1996].

An important point here, is that the change in the process of bidding resolution does not affect the buyers. It is an internal change of the system. This process is opaque to the buyers which cannot see the information passed between governors in order to resolve the bidding protocols. In other words, from the point of view of the buyers there is no difference between centralized bid resolution and the new scheme.

The buyers are the participants in the auctions and each one is connected to one governor in order to communicate with the other agents. Of course, we cannot specify the buyers here because each one is potentially unique. All we can define are the channels with which the buyer will be supplied in order to communicate with its governor, the messages that it will receive and when it is allowed to submit a bid or leave the system. Therefore, in the auction room there is one auctioneer, a set of buyers, and one governor for each buyer (see Figure 4.1). The auctioneer is connected to all the governors, each buyer is connected to a governor which is the only agent it can communicate with and

governors are structured in a ring. That is, apart from the auctioneer and its buyer a governor is connected to its predecesor and succesor in the ring.

We will focus on the specification of the distributed resolution of the bidding protocol. We omit the specification of the auctioneer but note that its principal work now is to send the information about the lots at the start of each round.

4.1.2 The π-calculus in brief

The main features of the π-calculus—and those necessary to read the remainder of this chapter—are the means to read and write information over channels, the creation of channels, and parallel, alternative and sequential composition. Terms in the π-calculus are described as prefixes followed by terms, which is intentionally a recursive definition. Syntactic details are outlined below[2].

Summary of π-calculus syntax and semantics

In this section we provide a short description of π-calculus. For more information, the interested reader is referred to Pierce's excellent article [Pierce, 1996] and subsequently to Milner [Milner, 1991]. The basic operations in π-calculus are:

$x(y)$: reads an object from channel x and associates it with the name y. This operation blocks until the writer is ready to transmit. The scope of y is limited to the process definition in which y occurs. Channel names, on the other hand may be local (see ν below), parameters to process definitions (see below), or global.

$\overline{x}\,\langle y\rangle$: writes the object named by y to the channel x. This operation blocks until the reader is ready to receive.

$\nu\, x\, ...$: creates a new channel named x. The scope of x is limited to the ν expression, but the channel may be passed over another channel for being used by another process. For example, a common idiom is to create a channel using ν, transmit it to another process and then wait for a reply on that channel:

$$\nu\,(x)\,\overline{y}\,\langle x, question\rangle\ .\ x(answer)$$

$P \mid Q$: the terms P and Q behave as if they are running in parallel. For example, $\overline{x}\,\langle 1\rangle \mid \overline{y}\,\langle 2\rangle$ outputs 1 on channel x and 2 on channel y simultaneously.

$P\ +\ Q$: either one or the other (non-deterministic choice) of P and Q proceeds. Normally, the prefixes of P and Q are operations which could block, such as channel transactions, and this operation allows us to express the idea

[2] A word of warning: this description should not be taken as definitive, since there are numerous interpretations which vary slightly in details of syntax, and sometimes of semantics. It does however represent π-calculus adequately for the purposes of the discussion here.

of waiting on several events and then proceeding to act upon one of them when it occurs. For example, $x(a) + y(a)$ waits for input on channels x and y, associating the information in both cases with a. As soon as one branch of such an alternative succeeds, the others can be considered to have aborted (see discussion in section 4.1.2).

$P \, . \, Q$: the actions of term P precede those of term Q. For example $x(y) \, . \, \overline{z} \, \langle y \rangle$ reads y from x then writes y on z.

In addition, we include an ability to associate a term with a name — that is a definition — and furthermore, by doing that a global channel is declared with that name, as in: $P(x1, x2, x3) = ...$, which defines a process P taking a three-tuple. In practice this also means we have declared a channel P such that we may invoke the process P by writing a three-tuple to the channel named P. We will use this convention to obtain a form of parameterization, allowing us to pass processes as arguments (high-order processes) by passing the channel by which they are invoked. This syntactic convenience can be described primitively in the π-calculus but we omit these details here.

Events and choice

Among the many variants of the π-calculus, we chose as a starting point, the basic synchronous form as found in [Milner, 1991]. One of the essential properties of the kinds of institutions we want to model is liveness, which in practical terms means an event-based model. The non-deterministic choice (sum) operator has therefore been invaluable—although it also raises some interesting questions. To quote [Milner, 1991]:

> The summation form $\sum \pi_i . P_i$ represents a process able to take part in one—but only one—of several alternatives for communication. The choice is not made by the process; it can never commit to one alternative until it occurs, and this occurrence precludes the other alternatives.

When viewed as a mathematical description, for example, for the purpose of determining bisimilarity, there is no problem. However, when viewed as a program to run, there is an element of time and therefore sequence involved. Consider the process $\overline{c_1}.P_1 + \overline{c_2}.P_2$. If a message arrives on c_1 just before one arrives on c_2, do we expect to become P_1, or do we expect a non-deterministic choice of P_1 or P_2? Certainly, we *can* become P_1, but most people (and the quote above can be interpreted to support this), would say we *should* become P_1. If not, then the π-calculus would be a difficult tool indeed, requiring many synchronizations to enforce this natural behaviour, and these synchronizations would generally have no counterpart in a “real” program. In the following descriptions we have assumed that the natural interpretation is the case, this is, choices are determined as and when messages arrive on channels.

$$
\begin{aligned}
&\mathsf{test/0}(event,\ then,\ else) = \\
&\quad \nu\ \ (c1, c2, c3) \\
&\qquad \overline{c3}\,\langle\rangle \\
&\qquad |\ event()\,.\,c2()\,.\,\overline{c1}\,\langle then\rangle\ +\ c2()\,.\,\overline{c1}\,\langle else\rangle \\
&\qquad |\ c3()\,.\,\overline{c2}\,\langle\rangle\,.\,c1(x)\,.\,\overline{x}\,\langle\rangle
\end{aligned}
$$

A further issue, of wanting prior further issue, of wanting prior, is addressed by the definition of the test/0 process, which is much used later on. This process is used to check if there is information waiting to be read on a channel but without blocking the process attempting to read.

The function tries to read from the channel *event* and if it succeeds, it writes on channel *then*, otherwise it writes on channel *else*. The version presented here does not read any data from the channel *event* but we assume that we have other versions that do, and writes it on channel *then*. We will differentiate the number of arguments passed by adding arity to the name of the process (following a Prolog convention). Then the function test/0 is the one that does not pass information, the process test/1 is the one that passes one item and so on.

4.1.3 Leader Election

Leader election is a distributed algorithm used in some kinds of networks to elect a leader. For example, it is used in a token ring network when the token is lost and it is necessary to generate a new one. The algorithm assumes that all the nodes are identical, except in each having an unique identifier and they have to select one node to generate the token, but only one because there can only be one token in the ring.

There are different versions for solving the Leader Election problem and we have based our work on the LCR version [Lynch, 1996]. This algorithm uses only unidirectional communication and does not rely on knowledge of the size of the ring. Other algorithms use more knowledge (equals more constraints) to reduce the complexity of the algorithm, but do not change it in essence.

It is presumed that the unique identifiers support an ordering so that the leader will be the process with the largest identifier. First of all, each node sends a message with its identifier around the ring. When a process receives a message there are three possible actions:

1. if the identifier in the message is greater than its own, it passes the message on.

2. if the identifier in the message is less than its own, it discards the message.

3. if the identifier in the message is equal to its own, it declares itself the leader.

Thus, only the process with the greatest identifier will receive again its message and it will declare itself as a leader. We can see that all the other messages will be eliminated because at some point they will arrive at a process with a greater identifier.

The important point of the algorithm that we have to bear in mind is that it only uses local information in each process and all of them are identical except in their identifiers. The processes do not have global information.

The next step is to see how we can apply this strategy to the auction protocols. The first and obvious point is that we have to compare the bids submitted by the buyers. The winner will be the buyer with the greatest bid. When a process receives a message, it will have to compare the bid in the message with its bid. There is one aspect that makes processing bids trickier than straight leader election: in classical leader election all the identifiers are different, but it is quite possible that there will be equal (highest) bids posted. Namely, there is a collision if there is more than one bid at the greatest price.

We have to define first when the messages are generated and what they should contain. When to generate is obvious, the governors will send a new message when the buyer makes a bid. For the second point is not enough just to send the bid: we need to know who has generated the message because it may happen that more than one buyer submits the same bid value and whether there are more buyers that have made a bid at the same value.

Thus, the messages will have two fields:

1. a list of the identifiers of the buyers that have bid at price *bid*,

2. the bid itself

From the list we can learn who has contributed to the message and if there is more than one buyer bidding at that price.

Now, we have to analyze what happens when a governor receives a message from its neighbour. When the bid in the message is different from its own, it acts as in the Leader Election algorithm: it passes the message on if the bid is greater and discards it if it is lower than its own.

The important point is what happens when it receives a message with a bid equal to its own. There are two possibilities:

1. *This is the message generated by itself* which implies that this governor's buyer has made the greatest bid. However, this is not enough to elect itself as a winner because another buyer could have made a bid at the same value. To distinguish this case, it has to look at the list of identifiers in the first field of the message, and if there is only its identifier, it can declare itself the winner. Otherwise there has been a collision, in which case it will generate a collision message in order to inform the other governors. We will explain later how collision messages are managed.

2. *Or, it is a message generated by another governor* which indicates that another buyer has made a bid at the same price and it *could* be a collision.

> This is not enough for declaring a collision yet because another governor, further round the ring, could have made a greater bid. It will only be a collision if this is the greatest bid. The problem is that the governor has only local information and it only knows that it *could* be a collision. The governor can only declare the collision when the message has made one complete round and if so, it is sure that this is the greatest bid. All it can do is to add its identifier to the list and pass on the message.

As we can see, no governor eliminates a message with a bid equal to its own. Thus, when there is a collision, this will be detected for all the governors involved in it and each one of them will generate a collision message. Thus, after a collision, there will be one collision message for each buyer involved in it. In the version that we propose here, the governors restart the round after a collision. Important remaining issues are, how collision messages are eliminated and how to ensure that each governor receives only one collision message. The solution is that each collision message travels over the part of the ring from the governor generating the message to the next governor involved in the collision. Thus, each governor will receive one and only one collision message.

Another point is what should a governor do when it knows that it has won a round. The answer is that it has to send a message around the ring in order to inform all the other governors that the round is finished. An issue to note is *what* information should be passed on to the buyers. They could just be notified of the end of the round, but more likely they could be sent some information about the result, such as the price and/or the identity of the winner. Precise choices depend on the conventions of the institution being modelled, but they are not important otherwise to the discussion here. This message synchronizes all the governors and subsequently the winner's governor sends a message to the auctioneer to inform it about the result of the round and also that the round is finished.

Now we have described all the possible cases when a governor receives a message. Hence, we can give a generic variant of the Leader Election for the resolution of bidding protocols (see Figure 4.2).

The last difficulty to address is what to do when no buyers make a bid. In this situation, no messages would be generated, so leading to deadlock, because each one of the governors will be waiting for messages. To avoid this, each governor is required to generate a message for each round *unless* its buyer has not submitted a bid. These messages have a bid value of the negation of the gpvernor identifier. Thus, if no buyer has submitted a bid, only the governor with the lowest identifier will receive its message back. It will then detect that there have been no bids and it will notify the auctioneer that the lot is withdrawn.

Before explaining the duties of the governor, we present an example in Figure 4.3. In this example there are three governors, two of them, governors G_1 and G_2, which bid 10 and another, governor G_2, which bids 8. The messages with bid 10 make the complete round while the message with bid 8 is eliminated as it reaches a governor that has made a greater bid. This provokes a collision which is detected by both governors when they received back their messages

1. if the governor receives an end of round message which it *did not* generate, it passes the message to the next governor.
2. if it receives an end of round message which it *did* generate, it eliminates the message and sends the result of the round to the auctioneer.
3. if it receives a collision message in which it *was not* involved, it passes the message to the next governor and restarts the round.
4. if it receives a collision message in which it *was* involved, it eliminates the message and restarts the round.
5. if it receives a message with a bid greater than its own, it passes the message to the next governor.
6. if it receives a message with a bid equal to its own and it *is not* its message it adds its identifier to the message and passes it to the next governor.
7. if it receives its own message back and it only contains its identifier in the first field, it is the winner of the round and it sends an end of round message.
8. if it receives its own message back but there is more than one identifier in the first field, there is a collision and it generates a collision message.
9. if it receives a message with a bid lower than its own, it eliminates the message.

Figure 4.2: Specification of the generic bidding resolution protocol

with more than one identifier. Then they generate a collision message, which has identifier -1. We can see that each governor receives only one of them. The governors which have participated in the collision wait until they receive another collision message. At that point, they eliminate the collision message and restart the round. The example finishes at that point but after the collision the governors will restart the round. That is to say, buyers will be informed about the collision and they will be requested to submit new bids.

In the next section we will give the specification of the governor in each protocol. We will explain how to apply the algorithm outlined above to each one and the modifications to take account of their individual characteristics.

4.1.4 The Governor

As we have said earlier, the governor has two important functions, handling communication between buyers and the institution (external) and resolving the bidding protocols (internal).

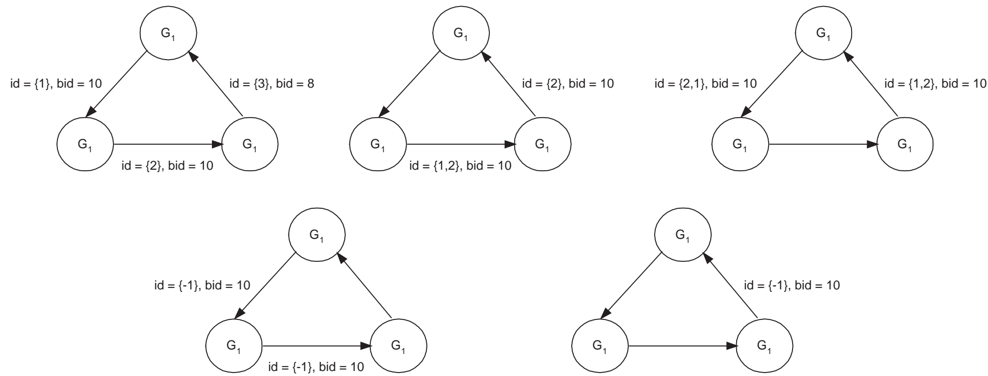


Figure 4.3: Example of collision resolution

The first function gives the buyer the communication infrastructure to enable participation in the auctions by passing to the buyer the information that it needs about the state of the auctions. For example, when a round starts, the characteristics of the lot offered, if he has won a round, etc.. Furthermore, it passes the buyer messages to the other institution agents. These are the bids of the buyer and when it wants to leave the system. It also checks that the messages of the buyer follow the protocol. For example, the buyer is not allowed to submit a bid at the wrong time. The idea is to abstract buyer developers from communication problems, allowing them to concentrate on bidding strategies.

The second function, which is independent of the buyers, consists of deciding who won a round or whether a lot is withdrawn, using the modified Leader Election protocol with some variations depending on the auction protocol. As we have said before, buyers are not allowed to see the information passed between governors in order to resolve a round. The governors have to be robust and have to incorporate security measures in order to protect them from malicious buyers. Otherwise, buyers could read the messages with the bids of the others buyers and then generate new bids out of sequence with the help of additional information. This is an important point to be borne in mind but here we focus just on the algorithm.

From the point of view of a governor, an auction round is divided in four steps as we can see in Figure 4.4.

1. **Start round:** This step corresponds to the period between two rounds. In that period, new buyers are added to the auction and existing participants can leave. The step finishes when the governor receives from the auctioneer the information of the next lot to be auctioned. This step is the same for each protocol.

2. **Waiting for bids:** This step can be seen as an initialization step. For each round it has a pre-determined time expressed in the lot information.

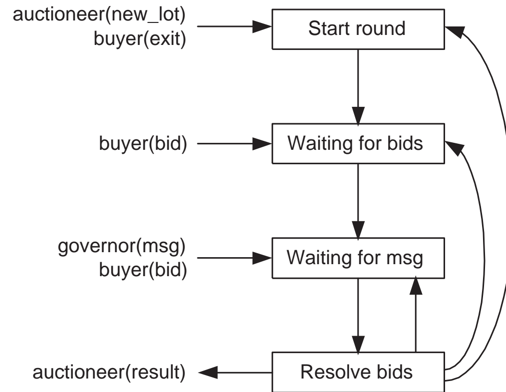


Figure 4.4: The four steps of a bidding round

The buyers have that time to make their first bid. Except in the English auction, this is the only period that buyers are allowed to submit bids. When this step is finished we can be sure that each governor has sent its neighbour a message. It is for that reason that we say that this is an initialization period.

3. **Waiting for next message:** In this step a governor waits for a message from its source governor (the previous one in the ring) or for a message from the buyer with a new bid. In the first case it passes to the next step in order to handle that message. If it receives a message for a new bid from the buyer it generates a new message and sends it to its destination governor (the next one in the ring). In the protocols that we specify here, only in the English auction are the buyers allowed to submit bids at that moment. In the other three they can only submit bids at the second step and this step only consists in waiting for a message from the source governor.

4. **Resolve bids:** This is the important part of the algorithm when a governor uses the cases explained before with the last message received and its information to decide what it has to do.

As we can see, the first step is common for all the protocols and the others present some differences between each one. The interesting thing is that all of them follow the same pattern unless they have their own characteristics. We will focus on these three steps for each protocol.

The first step receives the information of the next lot from the auctioneer and calls the WaitingBid function of the corresponding protocol. It is during this step when buyers can leave the auction and new buyers can join it.

Before presenting each protocol specification in π-calculus we define the channels used for communication between one governor and the other agents. These

are:

- *in*: reception of messages from its predecessor in the ring.
- *out*: transmission of messages to its successor in the ring.
- *b/gov_bid*: reception of the bids from the buyer.
- *gov/auc_res*: transmission of the result of the round to the auctioneer.
- *gov/b*: transmission of information to the buyer.
- *b/gov_exit*: reception of message from buyer with the desire of leaving the auction room.

Apart from the channels, the governor also keeps information in order to compare it with that in the incoming messages and the parameters of the current lot. Normally, in the first case it keeps the last bid submitted by the buyer — but in the English auction more information is needed as we will explain later. The information of the current lot is used after a collision for restarting the round.

Next, we give the specification of the protocols. We will begin with the complete specification of the First-Price/Sealed-Bid protocol, being the simplest. Then, we will explain the changes, by reference to that for the Vickrey and Dutch protocols, because they are similar to the first one. Finally, we will explain in more detail the last and more complex, the English auction protocol.

4.1.5 First-Price/Sealed-Bid

The main characteristic of this kind of auctions is that you can divide it in two phases. There is a time for submission of bids and afterwards, analyzing the bids it is decided the winner. That is to say, for each lot there is given a time for the buyers to submit their bids and after that the governors decide who is the winner. The winner will be the buyer who has submitted the greatest bid and this is the price that he will pay for the lot. The lots auctioned in this protocol are defined with one parameter which indicates the time that buyers have to submit bids. Next, we explain the last three steps for this protocol:

1. **Waiting for bids:** This step corresponds to the time that buyers have for submitting bids. In order to check if the buyer has submitted a bid in the given time it uses an auxiliary process called FP-bid. Before passing to the next step a message is generated whith the value returned from this process.

   The FP-bid process first waits for the time specified using the delay process (definition not given here) which waits for the units of time that it receives as a parameter. After that it uses the function test/1 in order to see if the buyer has made a bid. If it has, it returns the value of the bid, otherwise it returns the value $-id$ which indicates that the buyer has not submitted a bid.

$$
\begin{array}{l}
\mathsf{FP\text{-}WaitingBid}(id,\ in,\ out,\ b/gov_bid,\ gov/auc_res,\ gov/b,\ time) = \\
\quad \nu \quad (done, then, else) \\
\qquad \overline{gov/b}\ \langle time \rangle \\
\qquad .\ \ \mathsf{FP\text{-}bid}(b/gov_bid, time, done) \\
\qquad\quad |\ \ done(bid) \\
\qquad\qquad .\ \overline{out}\ \langle bid \rangle \\
\qquad\qquad .\ \mathsf{FP\text{-}WaitingMessage}(id,\ bid,\ in,\ out,\ b/gov_bid,\ gov/auc_res,\ gov/b, \\
\qquad\qquad\qquad time)
\end{array}
$$

$$
\begin{array}{l}
\mathsf{FP\text{-}bid}(id,\ b/gov_bid,\ time,\ done) = \\
\quad \nu \quad (then, else, done2) \\
\qquad \mathsf{delay}(time, done2) \\
\qquad |\ \ done2() \\
\qquad\quad .\ \ \mathsf{test/1}(b/gov_bid, then, else) \\
\qquad\qquad |\ then(bid)\ .\ \overline{done}\ \langle bid \rangle\ +\ else(junk)\ .\ \overline{done}\ \langle -id \rangle
\end{array}
$$

$$
\begin{array}{l}
\mathsf{FP\text{-}WaitingMessage}(id,\ bid,\ in,\ out,\ b/gov_bid,\ gov/auc_res,\ gov/b,\ time) = \\
\qquad in(id_i, n_bid) \\
\qquad .\ \mathsf{FP\text{-}ResolveBids}(id,\ bid,\ id_i,\ n_bid,\ in,\ out,\ b/gov_bid,\ gov/auc_res,\ gov/b, \\
\qquad\qquad time)
\end{array}
$$

Figure 4.5: Specification of the processes FP-WaitingBid, FP-bid and FP-WaitingMessage.

2. **Waiting for next message:** In this protocol during this step the governor only waits for a message from its predecessor in the ring because buyers are not allowed to submit multiple bids. After reading from channel *in* it sends a message to the process FP-ResolveBids for further processing.

3. **Resolve bids:** This is the most complex and interesting function. It applies the algorithm explained before with its own bid and the last message received. The algorithm in this case is exactly the same as explained above without changes.

In figures 4.5 and 4.6 we present the specification in π-calculus of the First-Price/Sealed-Bid protocol. Concretely in figure 4.5 it is presented the specification of the processes FP-WaitingBid, FP-bid and FP-WaitingMessage, while in figure 4.6 it is presented the specification of the process FP-ResolveBids.

4.1.6 Vickrey's auction

This protocol is very similar to the one before in the sense that it is also divided in two phases and with the same processes as the previous one. The sole difference is that the winner is the buyer who has submitted the highest bid *but* the price that he has to pay corresponds to the second highest bid. Then, messages are extended with a new field containing information about the second highest bid.

We do not give the specification because the only change from first-price is that the messages have one additional field. This field, corresponding to the second highest bid, is initialized to zero when a governor generates a message. Then, when a governor passes a message on, it has to compare this field with its bid and it updates it (if its own is higher). Then, the governor winnning the round sends the auctioneer the value of the new field. That is, the second highest bid because it is the price that the buyer has to pay for the lot.

4.1.7 Dutch auction

The main characteristic of this protocol is that it is a descending price auction. The round starts with a high price descending until one buyer submits a bid. In real Dutch auctions it is the auctioneer who calls out the offers until one buyer bids. Here, each governor sends independently the offers to its buyer [3]. While this confers several advantages in a distributed setting that cannot arise in the physical scenario, it does have the drawback that in contrast to a real auction, where when a buyer sees an offer he knows that no one has submitted a bid at a higher price, whereas here the buyer cannot be sure of this because each governor is running independently from the others. This could change the bidding strategy of the buyers.

The parameters of this kind of auction are slightly different, being:

- *start_price*: the starting price at which the governors start sending offers to the buyers.

[3] This technique and its justification are presented in [Padget and Bradford, 1999]

$$
\begin{aligned}
&\mathsf{FP\text{-}ResolveBids}(id,\ bid,\ id_i,\ n_bid,\ in,\ out,\ b/gov_bid,\ gov/auc_res,\ gov/b,\ time) = \\
&\begin{cases}
\textbf{if } car(id_i) = -2 \\
\begin{cases}
\textbf{if } n_bid = bid \\
\begin{cases}
\textbf{if } bid < 0 \\
\overline{gov/auc_res}\,\langle 0, 0\rangle \\
\textbf{otherwise} \\
\overline{gov/auc_res}\,\langle id, bid\rangle \mid \overline{gov/b}\,\langle \text{“}\texttt{Winner}\text{”}\rangle
\end{cases} \\
\textbf{otherwise} \\
\quad \overline{gov/b}\,\langle \text{“}\texttt{End of round}\text{”}\rangle \\
\quad \mid \overline{out}\,\langle -2, n_bid\rangle
\end{cases} \\
\textbf{elseif } car(id_i) = -1 \\
\begin{cases}
\textbf{if } n_bid = bid \\
\quad \overline{gov/b}\,\langle \text{“}\texttt{Collision}\text{”}\rangle \\
\quad \mid \mathsf{FP\text{-}WaitingBid}(id, in, out, b/gov_bid, gov/auc_res, gov/b, time) \\
\textbf{otherwise} \\
\quad \overline{gov/b}\,\langle \text{“}\texttt{Collision}\text{”}\rangle \\
\quad \mid \overline{out}\,\langle -1, n_bid\rangle \\
\quad \mid \mathsf{FP\text{-}WaitingBid}(id, in, out, b/gov_bid, gov/auc_res, gov/b, time)
\end{cases} \\
\textbf{otherwise} \\
\begin{cases}
\textbf{if } n_bid = bid \\
\begin{cases}
\textbf{if } car(id_i) = id \\
\begin{cases}
\textbf{if } |id_i| = 1 \\
\overline{out}\,\langle -2, bid\rangle \\
\textbf{otherwise} \\
\quad \overline{out}\,\langle -1, bid\rangle \\
\quad \mid \mathsf{FP\text{-}WaitingMessage}(id,\ bid,\ in,\ out,\ b/gov_bid,\ gov/auc_res, \\
\qquad gov/b,\ time)
\end{cases} \\
\textbf{otherwise} \\
\quad \overline{out}\,\langle id_i \cup \{id\}, bid\rangle \\
\quad \mid \mathsf{FP\text{-}WaitingMessage}(id,\ bid,\ in,\ out,\ b/gov_bid,\ gov/auc_res,\ gov/b, \\
\qquad time)
\end{cases} \\
\textbf{otherwise} \\
\begin{cases}
\textbf{if } n_bid > bid \\
\quad \overline{out}\,\langle id_i, n_bid\rangle \\
\quad \mid \mathsf{FP\text{-}WaitingMessage}(id,\ bid,\ in,\ out,\ b/gov_bid,\ gov/auc_res,\ gov/b, \\
\qquad time) \\
\textbf{otherwise} \\
\mathsf{FP\text{-}WaitingMessage}(id,\ bid,\ in,\ out,\ b/gov_bid,\ gov/auc_res,\ gov/b, \\
\qquad time)
\end{cases}
\end{cases}
\end{cases}
\end{aligned}
$$

Figure 4.6: Specification of the process FP-ResolveBids

$$
\begin{array}{l}
\mathsf{DA\text{-}bid}(id,\ b/gov_bid,\ actual_price,\ decrement,\ reserve_price,\ done) = \\
\quad \nu \quad (then, else, done2) \\
\quad\quad \overline{gov/bid}\,\langle actual_price\rangle \,.\, \mathsf{delay}(1, done2) \\
\quad\quad \mid \quad done2() \\
\quad\quad\quad .\ \ \mathsf{test/0}(b/gov_bid, then, else) \\
\quad\quad\quad\quad \mid \quad then() \,.\, \overline{done}\,\langle bid\rangle \\
\quad\quad\quad\quad\quad + \quad else() \\
\quad\quad\quad\quad\quad\quad . \left\{ \begin{array}{l}
\textbf{if } actual_price - decrement > reserve_price \\
\mathsf{DA\text{-}bid}(id,\ b/gov_bid,\ actual_price - decrement,\ decrement, \\
\quad\quad reserve_price,\ done) \\
\textbf{otherwise} \\
\overline{done}\,\langle -id\rangle
\end{array} \right.
\end{array}
$$

Figure 4.7: Specification of the process DA-bid

- *reserve_price*: the minimum price at which the lot may be sold. If that price is reached the buyer loses the opportunity to bid for the lot. If all the governors reach this price without a bid being made, then the lot is withdrawn.
- *decrement*: the difference between successive offers.

The process that is different from First-Price/Sealed-Bid auction is FP-bid. All the other processes are identical except that the *time* parameter is replaced by the three parameters presented above. The DA-bid process, specified in figure 4.7, sends an offer to the buyer, then waits one unit of time and checks if the buyer has submitted a bid. In that case it returns the actual price as the buyer bid, otherwise it decrements the value of the offer sent to the buyer and if the reserve price is not reached, it repeats the process again. This function stops when the buyer submits a bid or the *reserve_price* is reached.

4.1.8 English auction

This protocol is the most complex that we have specified and the one that presents the most differences with the others. Here bidding starts at a minimum price and the buyers submit increasing bids until all of them stop. Each buyer can submit as many bids as it wants before a winner is declared. In the real auctions the auctioneer says *going, going, gone* after a bid before declaring a winner. In order to model that here, before a governor either declares itself the winner or detects a collision, its message has to make three rounds over governors ring.

This protocol starts with a descending bidding protocol as in the Dutch auction until one buyer submits a bid. After that, the auction follows the pattern just described, with bids going up.

In order to count the laps of the message with the greatest bid received, each governor has to keep more information than in the previous protocols. It has

$$
\begin{array}{l}
\mathsf{EA\text{-}WaitingMessage}(id,\ bid,\ in,\ out,\ b/gov_bid,\ gov/auc_res,\ gov/b,\ l_id,\ l_bid,\\
\qquad count,\ start_price,\ decrement,\ reserve_price) = \\
\left\{
\begin{array}{l}
\textbf{if } l_bid < 0 \vee count = 2 \\
\ in(id_i, n_bid) \\
\ .\ \mathsf{EA\text{-}ResolveBid}(id,\ bid,\ id_i,\ n_bid,\ in,\ out,\ b/gov_bid,\ gov/auc_res,\ gov/b,\\
\qquad l_id,\ l_bid,\ count,\ start_price,\ decrement,\ reserve_price) \\
\textbf{otherwise} \\
\quad in(id_i, n_bid) \\
\quad .\ \mathsf{EA\text{-}ResolveBid}(id,\ bid,\ id_i,\ n_bid,\ in,\ out,\ b/gov_bid,\ gov/auc_res,\ gov/b,\\
\qquad l_id,\ l_bid,\ count,\ start_price,\ decrement,\ reserve_price) \\
\ + \quad b/gov_bid(n_bid) \\
\qquad .\ \overline{out}\ \langle \{id\}, n_bid \rangle \\
\qquad .\ \mathsf{EA\text{-}WaitingMessage}(id,\ n_bid,\ in,\ out,\ b/gov_bid,\ gov/auc_res,\ gov/b,\\
\qquad\qquad id,\ n_bid,\ 0,\ start_price,\ decrement,\ reserve_price)
\end{array}
\right.
\end{array}
$$

Figure 4.8: Specification of the process EA-WaitingMessage

to keep the last message that it has passed on, because it is the one with the greatest bid so far, and the number of times that it has received it in order to count the laps.

There is another important point in which it differs from real auctions. There are two situations where a buyer is not allowed to submit bids but where they can be allowed later. The first one is when the buyer has not made a bid at the waiting for bids step. After that, the buyer will be allowed to make bids if the governor receives a bid from another one. The second situation is when the governor receives a message for the third time. From the point of view of this governor the round is over. Although there is one possibility it may not be: if another buyer submits a greater bid before it has received the message three times. In the real auctions this does not happen because the auctioneer declares the end of the rounds in a centralized way. When the auctioneer declares that a lot is withdrawn or that there is a winner, there is no possibility of continuing the round.

So, when a governor receives a new message it has to compare it with the one that it has kept. If the bid is lower it discards the new message. If the bid in the new message is greater then it is kept, it sets the counter to zero and it passes the message on.

If it receives a message with a bid value that equals to the one it has been kept, different situations may arise. We have to bear in mind that there can be more than one message in the ring with the same bid value because no governor discards any message with a bid equal to its own. So, when a governor receives a message with a bid equal to the one in the message that has been kept, it compares the identifiers to see if they are the same message. If not, it passes the message on and waits for another one. If they are the same and the counter is less than two it passes the message on and increments the counter by one. If the counter equals two, it checks if it is its own message. In that case, it declares

EA-ResolveBid(id, bid, id_i, n_bid, in, out, b/gov_bid, gov/auc_res, gov/b, l_id, l_bid, $count$, $start_price$, $decrement$, $reserve_price$) =

if $car(id_i) = -2$
- **if** $n_bid = bid$
  - **if** $bid < 0$
    - $\overline{gov/auc_res}\,\langle 0, 0\rangle$
  - **otherwise**
    - $\overline{gov/auc_res}\,\langle id, bid\rangle \mid \overline{gov/b}\,\langle$“`Winner`”$\rangle$
- **otherwise**
  - $\overline{gov/b}\,\langle$“`End of round`”$\rangle$
  - $\mid \overline{out}\,\langle -2, n_bid\rangle$

elseif $car(id_i) = -1$
- **if** $n_bid = bid$
  - EA-WaitBid(id, in, out, b/gov_bid, $gov/auc\text{-}res$, gov/b, $start_price$, $decrement$, $reserve_price$)
- **otherwise**
  - $\overline{out}\,\langle -1, n_bid\rangle$
  - $\mid$ EA-WaitBid(id, in, out, b/gov_bid, $gov/auc\text{-}res$, gov/b, $start_price$, $decrement$, $reserve_price$)

elseif $car(id_i) = l_id \wedge n_bid = l_bid$
- **if** $n_bid > bid$
  - $\overline{out}\,\langle id_i, n_bid\rangle$
  - $\mid$ EA-WaitingMessage(id, bid, in, out, b/gov_bid, $gov/auc\text{-}res$, gov/b, l_id, l_bid, $count + 1$, $start_price$, $decrement$, $reserve_price$)
- **elseif** $count = 2$
  - **if** $|id_i| = 1$
    - $\overline{out}\,\langle -2, bid\rangle$
    - $\mid$ EA-WaitingMessage(id, bid, in, out, b/gov_bid, $gov/auc\text{-}res$, gov/b, l_id, l_bid, $count$, $start_price$, $decrement$, $reserve_price$)
  - **otherwise**
    - $\overline{out}\,\langle -1, bid\rangle$
    - $\mid$ EA-WaitingMessage(id, in, out, bid, b/gov_bid, $gov/auc\text{-}res$, gov/b, l_id, l_bid, $count$, $start_price$, $decrement$, $reserve_price$)
- **otherwise**
  - $\overline{out}\,\langle id_i, n_bid\rangle$
  - $\mid$ EA-WaitingMessage(id, bid, in, out, b/gov_bid, $gov/auc\text{-}res$, gov/b, l_id, l_bid, $count + 1$, $start_price$, $decrement$, $reserve_price$)

elseif $n_bid = l_bid \wedge count > 0$
- $\overline{out}\,\langle id_i, n_bid\rangle$
- . EA-WaitingMessage(id, bid, in, out, b/gov_bid, $gov/auc\text{-}res$, gov/b, l_id, l_bid, $count$, $start_price$, $decrement$, $reserve_price$)

elseif $n_bid > bid$
- $\overline{out}\,\langle id_i, n_bid\rangle$
- . EA-WaitingMessage(id, bid, in, out, b/gov_bid, $gov/auc\text{-}res$, gov/b, $car(id_i)$, n_bid, 0, $start_price$, $decrement$, $reserve_price$)

elseif $n_bid = bid$
- $\overline{out}\,\langle id_i \cup \{id\}, bid\rangle$
- . EA-WaitingMessage(id, bid, in, out, b/gov_bid, $gov/auc\text{-}res$, gov/b, l_id, l_bid, $count$, $start_price$, $decrement$, $reserve_price$)

otherwise
- EA-WaitingMessage(id, bid, in, out, b/gov_bid, $gov/auc\text{-}res$, gov/b, l_id, l_bid, $count$, $start_price$, $decrement$, $reserve_price$)

Figure 4.9: Specification of the process EA-ResolveBid

itself as a winner, if there is only its identifier and the bid is greater than zero. It declares the lot withdrawn if the bid is lower than zero, and a collision if there is more than one identifier in the message. If it is not its message, then it passes the message on but its buyer will not be allowed to make more bids unless it receives a new message with a greater bid as we have explained above.

One important point is that the identifiers in the list are always added at the end. So, in order to determine if two messages with the same bid are from the same buyer, it is enough to compare the first identifier of each one.

The last point to consider is that a buyer has always to submit a greater bid than the last that it received. Given this constraint, a governor can only receive a message that it has not generated with an equal bid to its own when its message is in the first lap around the ring. In this situation it has to add its identifier to the list in the first field of the message.

The first step is the same as in the Dutch auction, starting at one price and going down. Therefore, we will just specify the other two. In figure 4.8 it is presented the specification of the process EA-WaitingMessage, while in figure 4.9 it is shown the specification of the process EA-ResolveBid.

4.2 Conclusions

In this chapter, we have presented a distributed method for the resolution of the classic bidding protocols. For that purpose we have distributed the task from the auctioneer to the governors. Thus, the load of messages is also distributed because in the centralized versions all the messages go from the auctioneer to each governor or from each of them to the auctioneer. The specification given has been satisfactorily implemented using Pict [Pierce and Turner, 1997] , a π-calculus interpreter, giving some confidence in the validity of the method.

With the use of governors this very significant change can be done without affecting the buyers which can run without any knowledge of the way in which the bids are resolved. The governors also facilitate buyers communication allowing them to focus on bidding strategies.

We have given the specification of four protocols but we have defined the general steps in order to resolve bidding protocols in a distributed way. Thus it may simplify the specification of new auction protocols in a distributed manner. All what is necessary, is the definition of the three steps for the new protocol.

Another important point is that basing our algorithm on the Leader Election, we can use theoretical results established for Leader Election. This includes the algorithmic improvements mentioned in the introduction and, more important, its combination with techniques for termination detection and handling of stopping and Byzantine failure.

Chapter 5

Islander

In chapter 3 we have presented a formalisation of electronic institutions and we have also pointed out the importance of software tools that help institution designers to specify institutions. As a first step in this direction we have defined a textual specification language called ISLANDER based on the institution formalisation. This textual language which uses a Lisp like syntax, allows for a complete specification of all the components of an electronic institution. We believe that to textually specify an institution without any help is a really hard task. This is specially true for those elements which can be represented as a graph. Thus, we have developed the ISLANDER editor [de la Cruz, 2001, ISLANDER, URL, Esteva et al., 2002a] a tool for the specification and verification of electronic institutions. On the one hand, ISLANDER tries to make the work of the institution designer as easy as possible combining textual and graphical elements for the specification and on the other hand, it gives support to the verification of the specifications. This later point is crucial due to the complexity of these type of systems. The tool checks the correctness of the specifications before the engineer starts the development of the infrastructure for the institution.

The chapter is structured as follows: first in section 5.1 the ISLANDER language definition is presented; next in section 5.2, it is described the ISLANDER editor; and finally, in section 5.3, we focus on the verifications done by the tool.

5.1 ISLANDER language definition

In this section we present how the different components of an electronic institution, introduced in chapter 3, are specified in ISLANDER.

5.1.1 Electronic Institution

As we explained before an electronic institution is specified by a dialogic framework which defines the valid illocutions and the participant roles, a performative

structure which defines the activities within the institution, and a set of norms, that capture the consequences of agents' actions.

Thus, an electronic institution is defined in ISLANDER as follows[1]:

```
(define-institution institution-id as
  dialogic-framework = dialogic-framework-id
  performative-structure = performative-structure-id
  [norms = (norm-id+)]
)
```

In the next subsections we explain how these and the rest of the components in an electronic institution are specified in ISLANDER.

5.1.2 Dialogic Framework

The dialogic framework defines the valid illocutions that agents can exchange and the participant roles. In order to do so, it is defined which is the ontology, the content language used to encode the body of the messages, the set of valid illocutionary particles and the set of roles.

The roles that participating agents can play are divided into internal and external roles. The internal roles can only be played by the staff agents while external agents will only be allowed to play the external roles. Finally, relations over roles can be specified.

Thus, the definition of a dialogic framework contains the following elements:

- ontology: an identifier of a defined ontology that fixes which are the possible values for the concepts in a given domain.
- content-language: a language for the encoding of the knowledge to be exchanged among agents using the vocabulary offered by the ontology. Currently, only prolog and lisp are accepted as content languages.
- illocutionary-particles: a list of illocutionary particles to be used in the illocutions.
- external-roles: a list of roles that external agents may play.
- internal-roles: a list of roles that internal (staff) agents may play.
- social-structure: a list of triples of two roles and the relationship among them.

Next, we present the definition of the dialogic framework in ISLANDER:

[1] When an element is between brackets it means that it is optional, a '+' after an expression means 1 or more occurrences of the element, a '*' means from 0 to n occurrences of the element and elements between '{}' means that one of them must be choosen.

```
(define-dialogic-framework dialogic-framework-id as
  ontology = ontology-id
  content-language = {PROLOG,LISP}
  illocutionary-particles = (illocutionary-particle-id+)
  [external-roles = (role-id+)]
  [internal-roles = (role-id+)]
  [social-structure = ((role-id relation-id role-id)+)]
)
```

5.1.3 Ontology

The ontology defines the vocabulary that agents will use to exchange information. It defines what agents may talk about, fixing the concepts and their possible values in a concrete domain. The ontology is defined as a list of external types (which are defined somewhere else), a list of data type definitions and a list of function definitions. On the one hand, data types are defined by a name for the new defined data type, a constructor and a list of types. On the other hand, functions are defined by a name for the new defined function, a list of types representing the types of the function parameters and the retuning type. Functions returning a boolean can be considered as predicates where the type of the function parameters define the type of the predicate terms. Then, these predicates can be used to express the illocutions' content. As we pointed out, in chapter 3 we assume the following basic types: numeric, string and boolean. In the case of the numeric type in ISLANDER we allow to differentiate between integer (noted as int) and float. A special type is the agent identifier type, noted as *AgentId*, which is a subtype of string. The valid values for a variable of this type will be the identifier of an agent taking part in the scene. Also lists of any basic type or defined type can be used on the ontology definition.

Then, an ontology is specified in ISLANDER as follows:

```
(define-ontology ontology-id as
    {(type type-id),
     (datatype type-name = constructor-id of type-expression),
     (function-id : type-expression -> type-id)}+
)

type-expression ::= type-id | type-id * type-expression |
                    type-id list |  type-id list * type-expression
```

As the same illocution scheme can be used in different arcs of the same or different scenes, we allow to define illocution schemes out of the context of a scene. Thus, defined illocution schemes can be used to label different scene arcs by labelling the arc with its identifier. Since we want the scene protocol to be independent of concrete agents and time instants, scene arcs must be labelled with illocution schemes where at least the terms referring to agents and time are constants. Then, it is only allowed to specify illocution schemes

satisfying this restriction. The specification of an illocution scheme defines the illocutinary particle, the sender, the receiver, which can be a concrete agent, all the agents playing a role or all the agents within a scene, the content of the illocution specified as an expression in a content language and an optional time variable. As previously mentioned, the allowed content languages are Prolog and Lisp. If the content language is Prolog, the content is specified by the name of the predicate and then, the predicate terms written between parenthesis and separated by commas. If the content language is Lisp, the content is written between parenthesis, first it is written the name of the predicate and then the predicate terms separated by blanks. The arguments can be constants of the type of the parameter, variables, which must start by the symbols '?' or '!', or data type definitions if the parameter has been declared of a data type defined in the ontology.

For instance, imagine an ontology which contains the following function and data type definitions: $bid : string \times price \rightarrow boolean$ and $datatype\, price = Price\ of\ int$. Then, a valid content language expression in Prolog is $bid(?good_id, ?offer)$. The same expression in Lisp is written as $(bid\ ?good_id\ ?offer)$. We can also use a value from a data type definition in the second parameter, as for instance in the following expression in lisp, $(bid\ ?good_id\ (Price\ ?offer))$. A difference between the last two expressions is that in the first expression variable *offer* is of type *price*, while in the second expression it is of type *int*.

Notice that defined illocution schemas will be verified in the context of the scenes where they will be used. That is to say, each defined illocution scheme will be verified taking into account the dialogic framework and the ontology of the scene where it is used. Illocution schemes are specified in ISLANDER as follows:

```
(define-illocution-scheme illocution-scheme-id as
  illocutionary-particle = illocutionary-particle-id
  sender = (agent-var {role-id, role-var})
  receiver = {(agent-var {role-id, role-var}), all, role-id}
  content = content-language-expr
  [time = time-var]
)

content-language-expr   ::= {prolog-expr, lisp-expr}
agent-var ::= {?id, !id}
role-var  ::= {?id, !id}
time-var  ::=  ?id
```

5.1.4 Performative Structure

A performative structure defines the activities (conversations) that can take place within an institution and how agents depending on their role can move among them. As mentioned, it can be seen as a network of scenes mediated

by transitions. Then, the specification of a performive structure in ISLANDER contains the definition of the graph nodes, that is the scenes and transitions that compose the performative structure, and the connections among them. Scenes and transitions are specified by an identifier within the performative structure and by their class. This permits to have more than one scene and transition of the same class within the same performative structure. In the case of scenes, the class corresponds to a specified scene pattern which determines the scene protocol and the roles that can take part in it. Furthermore, it can be specified if multiple instances of the same scene running simultaneously will be allowed at execution time. In the case of transitions, they can be either of one of the transition classes, which are: *And* and *Or*.

The labels of the connections determine how agents, depending on their role, can move among the different scenes and transitions. Labels are expressed as conjunctions and disjunctions of pairs formed by an agent variable and a role identifier. A conjunction of pairs of an agent variable and a role identifier means that agents playing these roles must progress together through the arc, while a disjunction means that they can progress independently. As we have said, the labels of the arcs must be expressed in a disjunctive normal form. This is specified as a list of lists of pairs of an agent variable and a role identifier. Thus, each sublist defines a set of roles that can progress together through the arc. Furthermore, the arcs connecting scenes and transitions might have some constraints associated that must be satisfied by the agents in order to progress through the arcs. As pointed out in chapter 3, the basic elements to construct the constraints are illocution schemes and obligation predicates. Furtermore, it can be expressed the negation, conjunctions or disjunctions of them. On the other hand, the arcs connecting transitions and scenes determine if agents will incorporate to *one*, *some*, or *all* the current executions of the target scene or if agents will be incorporated to a *newly* created execution of the target scene. Finally, from the set of scenes, it must be selected which will be the initial scene, that will be the enter point of the institution, and the final scene which will be the exit point.

Summarising, the definition of a performative structure contains the following elements:

- scenes: list of the scenes of the performative structure containing for each scene its name and its class. If there can be multiple instantiations of a scene, this will be denoted by the word 'list' after the class name.

- transitions: list of the transitions of the performative structure containing for each transition its name and its class.

- connections: list containing the connections from scenes to transitions and from transitions to scenes. In the first case, the connection is expressed by the source scene, the target transition, a list of lists of pairs of an agent variable and a role identifier, and a list of constraints that will restrict agents' movements. In the second case it is expressed by the source transition, the target scene, a list of lists of pairs of an agent variable and a

role identifier, and the arc type. The arc type defines if a new execution of the target scene will be created or if the agent(s) will go to one, some or all current executions of the target scene.

- initial-scene: the initial scene – from one of those given in **scenes**.
- final-scene: the final scene – from one of those given in **scenes**.

Then, a performative structure is defined in ISLANDER as follows:

```
(define-performative-structure performative-structure-id as
  scenes = ((scene-id  scene-type-id [list])+)
  transitions = ((transition-id transition-type-id)+)
  connections =
   ({(scene-id transition-id (((agent-var  role-id)+)+) [arc-constraints]),
     (transition-id scene-id ((agent-var  role-id)+)+) destination)}+)
  initial-scene = scene-id
  final-scene = scene-id
)

transition-type ::= {And,Or}

destination ::= {new,one,some,all}

arc-constraints ::=  (not arc-constraint) |
                     (and arc-constraint arc-contraint) |
                     (or arc-constraint arc-constraint).|
                     obl-predicate |
                     illocution-scheme

illocution-scheme ::=
   {illocution-scheme-id,
    (illocutionary-particle-id (agent-var {role-id, role-var})
     {(agent-var {role-id, role-var}), all, role-id}
     content-language-expr [time-var]}

obl-predicate =  (obl agent-var illocution-scheme scene-id)
```

5.1.5 Scene

A scene defines an interaction protocol for a group of agents. In order to define a scene it is necessary to specify which are the participant roles, the dialogic framework which determines the illocution schemes that can label scene arcs, and the conversation protocol. The conversation protocol is specified by a directed

graph. The graph is specified by its nodes, which represent the conversation states and the connections among them labelled with illocution schemes and constraints, or timeouts. Furthermore, there is specified an initial state and a set of final states.

For those arcs labelled with illocution schemes, the user can choose between actually defining the illocution schema when the connection is defined or making reference to a previously defined illocution scheme labelling the arc by its identifier. As we explained above, in the illocution schemes labelling the arcs at least the terms referring to agents and time must be variables; the rest of the terms can be either variables or constants. Remember that we differentiate when an occurrence of a variable is to be bound and when an occurrence of a variable is to match its last bound value. When a variable is prefixed by the symbol '?', it is a binding occurence and a new binding will be created for the variable with the value in the uttered illocution. When a variable is prefixed by symbol '!', it is an application occurrence and the value on the uttered illocution must be the last bound value of the variable.

Arcs labelled with illocution schemes may have some constraints attached to them. Constraints can restrict the valid values of illocution schemes' variables and the paths that the scene evolution can follow, and are specified as boolean expressions by an operator and two expressions as follows: (*op expr expr*). In section 3.2.2 we have defined the operators that can be used within the constraints and expressions can be formed as Lisp expressions.

The basic elements to define constraints expressions are constants, scene variables and lists. For this latter case, remember that when a scene is executed the bindings of the variables for uttered illocutions are kept. Then, different sets of bindings for a variable can be obtained. Concretely, we express it, in the most general case, as $!^{i}_{w_i w_j} x(cond)$ which returns all the bindings of variable x in the i last sub-dialogues between w_i and w_j such that the substitution where the binding appears satisfies the condition *cond*. This is expressed in ISLANDER as $(!\ x\ w_i\ w_j\ i\ (cond))$.

Time-outs are specified as numeric expressions using numeric constants and numeric variables. In order to avoid the possibility of having more than one arc labelled with the same time-out value, from each state it can only be one outgoing arc labelled with a timeout is permitted. Otherwise, it may provoke an indeterminism in the path that the scene evolution must follow, as time-outs may contain scene variables and their bound values can not be determined at specification time. Alternatively, a time-out can be defined as the minimum of a set of numeric expressions. Then, each time the state is reached all the expressions are evaluated and the value of the timeout will be the minimum value of the expressions.

In order to specify at which points agents can join and leave a scene, for each role of the scene a set of access and exit states are defined which determine respectively when agents playing that role can join or leave the scene. Also, the minimum and maximum number of agents that can participate simultaneously within the scene playing each role is defined. This last field is optional and if

nothing it is said about the minimum or the maximum of a role it is understood that its minimum is zero and that there is no limit on the number of agents that can play the role within the scene.

Summarising, the definition of a scene contains the following elements:

- roles: list of roles that may participate in the scene.
- dialogic-framework: the dialogic framework to be used for communication within the scene.
- states: list of the states of the conversation graph.
- initial-state: the initial state.
- final-states: list of final states.
- access-states: list of pairs of a role identifier and a list of states, identifying which roles may join at which states.
- exit-states: list of pairs of a role identifier and a list of states, identifying which roles may leave at which states.
- agents-per-role: list of triples of a role identifier, minimum and maximum, defining the constraints on the population of a particular role. If nothing is said about the minimum or maximum for a concrete role, it is understood that the minimum is zero and that there is no maximum for this role.
- connections: list of transitions between scene states. Each one comprises one or a list of source state, a succeeding state, and either an *illocution-scheme* with a list of constraints over scene variables, expressed by an operator and two expressions, which must be satisfied to progress through this arc or a timeout that will trigger the transition when expired..

Next you can see the definition of a scene in Islander:

```
(define-scene scene-class-id as
 roles = (role-id+)
 scene-dialogic-framework = dialogic-framework-id
 states = (state-id+)
 initial-state = state-id
 final-states = (state-id+)
 acces-states = ((role (state-id+))+)
 exit-states = ((role (state-id+))+)
 [agents-per-role = (([min <=] role-id [<= max])+)]
 [connections =
  (({state-id,(state-id+)} state-id {illocution-scheme
           ((op left-expr right-expr)*), time-out})+)]
)
```

```
illocution-scheme ::=
   {illocution-scheme-id,
    (illocutionary-particle-id (agent-var {role-id, role-var})
     {(agent-var {role-id, role-var}), all, role-id}
     content-language-expr [time-var]}

time-out ::= (min numeric-expressions-list) | numeric-expression

op :: = { <, >, !=, <=, >=, <>, in, notin, subset, subseteq, or}
```

5.1.6 Norms

Norms capture the consequences of agents' actions within institutions. These consequences are expressed as obligations that agents will acquire or satisfy depending on their actions within the different scenes. The actions are expressed as pairs of illocution scheme and scene where the illocution must be uttered. Norms are specified in the following form: the actions that provoke the activation of the norm and restrictions over illocution scheme variables expressed in the antecedent, the actions that agents must carry out in order to fulfil the obligations expressed in the defeasible antecedent, and the set of obligations expressed on the consequent. The antecedent defines the set of illocutions that when uttered in the corresponding scene satisfying the boolean expressions will trigger the norm making the set of obligations expressed in the consequent hold. The defeasible antecedent defines the illocutions that must be uttered in the defined scenes in order to fulfil the obligations.

Then the definition of a norm contains:

- antecedent: a list comprising an arbitrary number of pairs of scene and illocution-scheme and a list of boolean expressions over illocution scheme variables.

- defeasible-antecedent: a list comprising an arbitrary number of pairs of scene and illocution-scheme.

- consequent: a list of obl predicate(s).

Norms are defined in Islander as follows:

```
(define-norm norm-id as
  antecedent = {((scene-id illocution-scheme)+),
                ((scene-id illocution-scheme)+) ((bool-expr)+)}
  defeasible-antecedent = {((scene-id illocution-scheme)+)}
  consequent = {((obl agent-var illocution-scheme scene-id)+)}
)
```

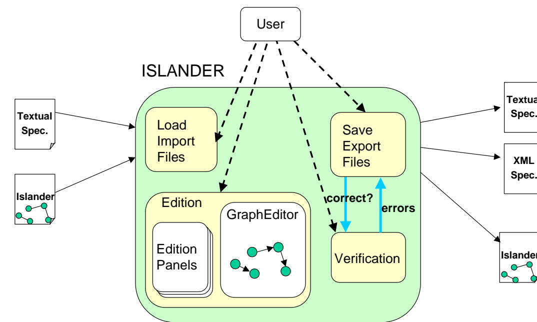


Figure 5.1: ISLANDER editor modules

5.2 ISLANDER editor

The ISLANDER language presented above, allows to specify all the elements of an electronic institution, but we think that it would be very hard for the designer of an institution to specify all its components textually. This is specially true for those elements represented as a graph. This is the case for the conversation protocol in scenes, the relationship among roles in the dialogic framework, and the performative structure. It is obvious that humans define and understand better a graph in its graphical representation than as text. Thus, we decided to develop an ISLANDER editor which combines graphical and textual specification facilitating the designers work. One of the important parts of the application is precisaly the graph editor that allows to specify the elements mentioned above. One of the outputs of the tool is the specification of the institution in the textual language presented in the previous section..

Apart from permiting institutions' specification the tool permits the verification of the specifications. Due to the complexity of this type of systems we believe that the verification of specifications is fundamental before deploying the system. In section 5.3 we focus on the properties verified by the tool and we explain how the verification of these properties is done.

5.2.1 Islander editor modules

The ISLANDER editor [Esteva et al., 2002a, ISLANDER, URL, de la Cruz, 2001] is divided in four main modules as we can see in figure 5.1: Import/Load, Export/Save, Edit and Verification. Next we describe each of the modules. Also in figure 5.2 we can see the Data Flow Diagram of the application.

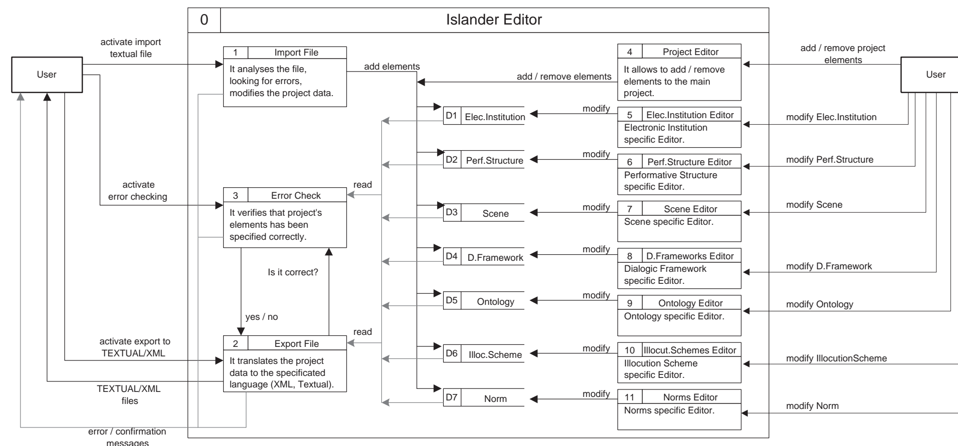


Figure 5.2: ISLANDER Data Flow Diagram

File Management

In this subsection we explain the Load/Import and Save/Export modules. ISLANDER editor can certainly save the current specification and load previous specifications. But apart from its own format files, ISLANDER can import and export other formats as we can see in figure 5.1. The principal difference is that the application files contain graphical information of those elements specified as a graph while the other type of files only have a textual representation of the graph. That is to say, the application files contain all the necessary information to draw the graphs.

ISLANDER can import textual specifications. When a textual specification file is imported it goes through the verification process which informs the user of any error found. Moreover, for those elements with graphical representation this representation is automatically generated allowing the user to interact with it afterwards.

The ISLANDER editor permits to export two types of files: textual and XML. On the one hand it permits to export the institution specified in the previoulsy given language. On the other hand, it permits to export the specified institution in XML. Before generating any of these type of files the specification goes through the verification process to check if it contains any error. If this is the case, the user is informed, and it can choose if she wants anyway to generate the file or not. This is important because the exported files can be used in subsequent steps of the development of the specified institutions.

Edition of electronic institutions

The edition module is the main module of the application as it permits users to add, modify or remove all kind of elements of an institution specification. Each element is specified in a different way depending on its characteristics.

Thus, ontologies, illocutions and norms are specified textually. On the contrary, dialogic frameworks, performative structures and scenes combine textual and graphical components in their specification. As in any edition process, the user interface is key in order to make as easy as possible the work of the institution designer. In section 5.2.2 we explain in detail the graphic user interface of the application.

One of the important parts of the edition is the graph editor used to specify the relationship between roles in the dialogic framework, the scene protocol and the performative structure. After studying our requirements and the available tools, we decided to develop our own graph editor. Thus, we have developed a graph editor that fulfils our requirements in the type of nodes, connections and type and number of labels per arc. The graph editor has been developed independently from the rest of the application. Thus, it can be updated without having to change the components using it and it can be used in any other application that needs a graph editor.

In our case, the graph editor is used by the components editing the dialogic framework, the scene and the performative structure. The graph editor does not have any information neither about the semantics of the graph which is representing nor about which restrictions this imposes on the topology of the graph. It has only information about which are the nodes of the graph, the type of each node, the connections between them and the labels of the arcs. Then, when the user wants to perform an operation over the graph the graph editor informs the involved element and it is the element who authorises or denies the operation. For instance, in the case of a performative structure we do not allow a connection between two transitions or two scenes. Then, when a user tries to create a new connection the graph editor informs the performative structure which will deny the new connection if the user is trying to connect two transitions or two scenes. Otherwise it will authorise the new connection. When an operation over the graph has been done the graph editor communicates it to the application interface that updates the information shown to the user.

The rest of the elements within an institution are specified textually. The tool tries to structure the way in which the textual information is entered, to make the work of the designer easier. Thus, for each element the required information is divided into different fields that have to be filled in and each field is labelled by a name identifying which information it contains. For instance, when the user has to specify an illocution scheme it is required to fill in a set of fields corresponding to the illocutionary particle, the sender, the receiver, the content expression and the time variable. We believe that it is easier to enter the information in this way that with a unique textual field where the user must enter the illocution scheme in its ISLANDER textual representation. Whenever it is possible, pop-down menus are used. This is used for fields that contain references to other specification elements and for fields whose value is one of a predefined set. This facilitates the designer work because he only has to select the element from a list and it also reduces typing errors. Anyway, for those elements which are a reference to another element the user can always opt for choosing one of the

list or introduce a new value for the field. This permits the user to specify the different elements in the prefered order.

Verification module

The verification module verifies whether the specification is correct. This is a fundamental part of the application. As open multi-agent systems are very complex it becomes crucial to be able to verify their specifications. Some verifications can be done while the user is editing the specification, but some of them need to wait until the specification is finished. Moreover, this allows the user to define the elements of the institution in the order that she wants. For example, she can make references to elements not yet defined without being constantly interrupted by error messages. The user can activate the verification of the current specification whenever she wants. Also, as we have mentioned above, when she wants to import a textual specification or to export textual or XML versions of the current specification the verification process is run in order to inform her about possible errors.

We have to take into account that to debug and find errors in a distributed system is a really hard work. Then, to verify the specification and detect the errors before starting the development of the system is fundamental. This saves a lot of time and effort from the developers of the system.

The tool has to verify that specifications satisfy different conditions. As we can see from the definition of the textual language some of the fields correspond to references to other specification elements. Then, it is obvious that the tool must check that each element which is referenced is actually defined. But there are other properties that must be checked. It must be checked that agents can reach the different scenes and that from each of them there exists a path that will allow them to leave the instittution, that scene protocols are correct and that agents may fulfil the norms. In section 5.3 we will focus on the verifications done by the tool.

5.2.2 Graphic User Interface

The graphic user interface is an essential aspect of the application because it is the way in which the institution designer interacts with the tool. In this sense the interface has been developed as user friendly as possible. We can see in figure 5.3 how the interface is divided in six panels that we explain next:

1. Menus and tool bar.

The menus contain the general operations of the application and they are similar to other applications. In the file menu the options are: new project, open project, re-open project, close project, save, import a textual specification, export the textual representation, export the XML representation, and exit.

In the insert menu the user can insert any type of element of an institution: a performative structure, a scene, a dialogic framework, an ontology, an illocution scheme and a norm.

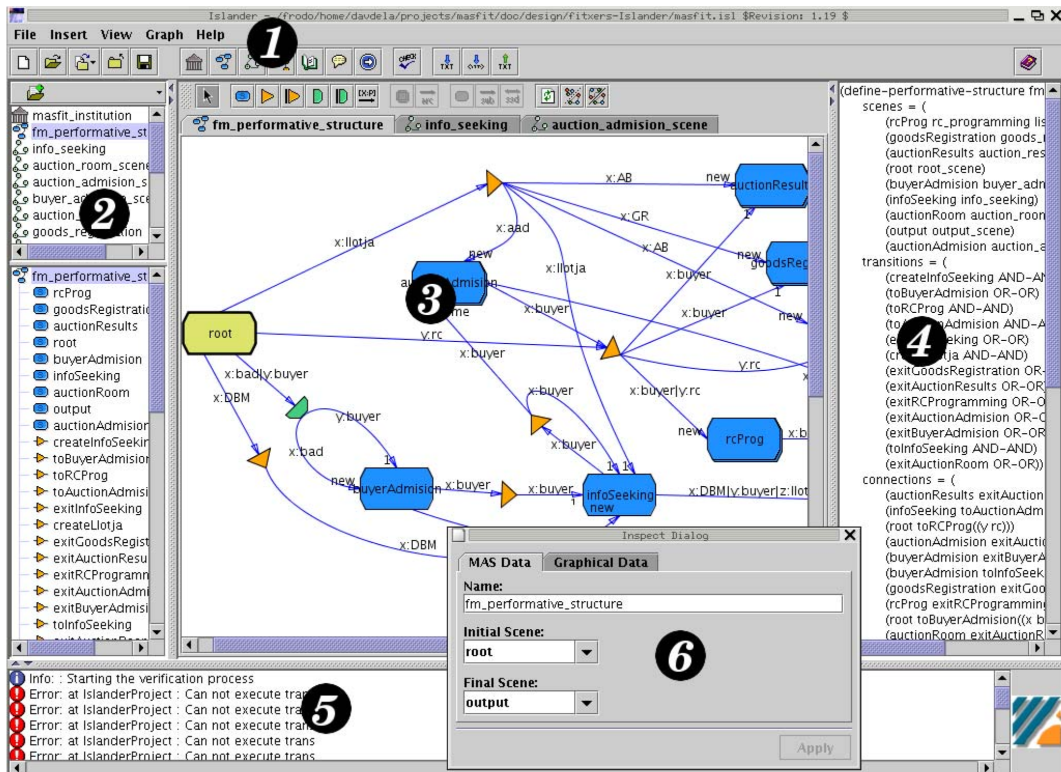


Figure 5.3: ISLANDER Graphic User Interface

THe view menu permits to show and hide the inspect panel, while from the help menu the user can activate the help of the application. The help contains a brief description of the application.

The tool bar contains icons for a quick access to the different operations. An important one is the icon that activates the verification process.

2. Project Structure Panel.

In this panel the user can see on the upper part all the elements that belong to current specification ordered by category. On the lower part the sub-elements of the currently selected element are displayed. For instance, when a performative structure is selected in the upper part of the panel, on the lower part the scenes, transitions and connections among them, which compose the selected performative structure are shown. Using these two parts of the panel the user can navigate through the different elements and sub-elements of the current specification. When she changes the selection the other panels are modified appropriately in order to show the information of the selected element or sub-element.

3. Graph Panel.

The graph panel supports the edition of the graphical components of the electronic institution. That is to say, the edition of the graphical component of

performative structures, dialogic frameworks and scenes. The user can edit the different graphs and modify them using the mouse. The graph panel is used for the creation and modification of the graph topology while the textual information associated to the graph is introduced and modified using the inspect panel. For instance, the graph panel is used to add a new node on the graph but the name of the node is introduced using the inspect panel. On the upper part there are icons that permit her to change from one graph to another. Also there is a tool bar with icons that are used to select the edition mode determining if the next action will be the selection of an element, to add a node of the selected type, or to add a connection between two nodes of the graph. As we will explain later the representation of the graph can also be modified textually from the inspect panel.

4. Textual Data Panel.
The textual data panel presents the textual representation of the current specification. Concretely, it presents the textual representation of the currently selected element of the specification. This permits the user to see how the current specification is represented in the presented textual specification language. This panel is only used for showing the textual representation of the current specification as the user is not allowed to modify the specification using this panel. Modifications on the specification must be done using the graph and inspect panels. We want to note that any change done by the user in the specification is immediately reflexed in the textual representation showed in this panel.

5. Log Panel.
The log panel is used to inform the user of the errors found in the specification. Some textual information is checked when the user modifies it, but the most important use of the panel is after running the verification process. Then, all the error found are presented to the user in the log panel. Each error message contains the element, and when possible also the sub-element, where the error was found and a message explaining the error. For each error the user can move to the element containing the error by simply selecting it. When a user selects an error the other panels of the application are modified in order to show her the element or sub-element containing the error. This permits the user to move faster to the element containing the error and to found it, facilitating her work.

6. Inspect Panel.
This panel is used for the definition of textual components of the specification. It allows the user to modify the attributes of each one of the elements in the specification. Whenever possible it uses pop-down menus to facilitate the designers work. This panel is always presented in front of the others and it always contains the information of the currently selected element. For each element a different design of the panel has been done because each one has different attributes. In any case we have tried to structure the information required in a way that permits a faster identification of the meaning of the fields to be filled in. This is also useful when specifications are presented to people not familiarised with the concepts of electronic institutions and their scpecification.

As soon as the user modifies an attribute of the selected element the rest

of the panels are updated in order to maintain all the information consistent. When the selected element corresponds to a graph element this panel also allows to modify its graphical attributes.

5.3 Verification

In the following we concentrate on the properties that the tool has to verify. We have divided the verifications in four parts: integrity, liveness, protocol correctness, and norm correctness. The first one checks principally that cross references among ISLANDER elements are correct. The second one checks that agents will not be blocked at any point of the performative structure, that each scene is reachable for each of its roles and that from each scene they can always reach the final scene. The third one checks that scene protocols are correct and finally, the last one checks that agents can fulfil the norms.

5.3.1 Integrity

As it can be seen from the definition of the textual specification language some of the fields correspond to references to other specification elements. It is obvious that the tool must check that each element which is referenced is actually defined. For instance, if it is defined that a scene uses a dialogic framework, this dialogic framework must be specified. Moreover, it must be checked that identifiers of the same type of element in the same context are not repeated. That is to say, that there can not be two scene definitions with the same identifier or within a scene definition there can not be two states with the same name.

Another important point is that within an institution the dialogic framework and ontology of each scene must be a subset of the institution dialogic framework and ontology respectively. For instance, it can not be the case that a dialogic framework of a scene has declared a role which is not declared in the institution dialogic framework. In the case of ontologies it is checked that the ontology referenced in the scene dialogic framework is a subset of the ontology referenced in the institution dialogic framework. That is to say, the scene ontology does not include any type, data type or function definition which is not defined in the institution ontology, and data types and functions are defined as in the institution ontology. Thus, it is checked that for each scene type used in the performative structure its dialogic framework and ontology are a subset of the institution dialogic framework and ontology respectively.

5.3.2 Liveness

An important part of the verification is related to the mobility of participating agents within the electronic institution. That is to say, that participating agents will not be blocked indefinetly at any point of the performative structure, that the different scenes can be reached by agents playing the scene roles and that

from any point of the perforamtive structure an agent has a path that will allow it to leave the institution. Thus, the following properties must be guaranteed:

- for each scene of the performative structure and for each role of the scene there is a path from the initial scene which will allow agents to reach the scene with that role.
- from each scene of the performative structure and for each role that can be played within the scene there is a path that permits agents playing that role to leave the institution; i.e. there is a path to the final scene.
- for each transition of the perfomative structure each agent reaching the transition has at least one path to follow.
- for each role of each scene agents playing that role are allowed to enter and leave.
- for each scene there is at least one incoming arc of type *new* that will allow the creation of scene executions.
- for each set of roles appearing together in a conjunction in an outgoing arc of a scene there is at least a scene state which is an exit state for all of them.
- for each set of roles appearing together in a conjunction in an incomiing arc of a scene there is at least a scene state which is an exit state for all of them.

We can see that these properties involve verifications on the performative structure and on the scenes composing it. On the one hand, the performative structure defines how agents depending on their role can move among the different scenes. On the other hand, scene definitions determine whether agents depending on their role will be allowed to join or leave the scene.

The first property is related to scenes' *accessibility*. That is, that each scene is accessible from the initial scene for all of its roles. The second property forces the final scene to be reachable from any scene of the performative structure, allowing agents to leave the institution. Summarising, for each scene of the performative structure and for each role that can be played within it, there must be a path connecting the initial and final scenes that passes through the scene. Notice, that only the roles of a scene can appear in the labels of its incoming and outgoing arcs. On the one hand, each role of a scene must appear in at least one label of its incoming arcs in order to allow agents playing that role to reach the scene. On the other hand, in order to allow agents to leave the scene, each of its roles has to appear in at least one label of its outgoing arcs. Thus, agents playing the participant roles can reach and leave the scene. Furthermore, each scene, except the initial and final scenes, must have at least one incoming arc of type *new*. At the begining of an institution execution only an execution of the initial and final scenes are created, while scene executions of the rest of the scenes are created

as agents progress through an arc of type *new*. Then, if there is a scene which does not have any arc of type new, this will imply that no scene executions of that scene will be created at execution time.

We also have to verify that agents will not be blocked at the transitions. That is to say, agents reaching a transition must have at least one path to follow in order to leave it. Performative structure arcs are labelled with pairs of an agent variable and role identifier which determine which agents depending on their role can move through the arc. If an agent playing role R_i reaches a transition following an arc which contains the pair $(x\ R_i)$ in its label, then its possible paths from the transitions are the outgoing arcs containing in its label a pair $(x\ R_j)$. Remember that we allow to change agents' role when traversing a transition. Summarising, the paths that an agent reaching a transition can follow is determined by the agent variables appearing on its outgoing arcs. Which of this paths the agent will follow will be determined by the type of the transition, but this is not important at this stage as we only want to guarantee that agents will not be blocked at the transition. Thus, it is checked that all agent variables labelling an incoming arc of a given transition appear on at least one label of its outgoing arcs. Furthermore, it is also checked that all agent variables appearing in the labels of outgoing arcs appear in a label of the incoming arcs because it has no sense to have an agent variable labelling only an outgoing arc of a transition. Summarising, it is checked that the set of agent variables appearing in the incoming and outgoing arcs of a transition are the same.

How are these properties checked? The two first properties are checked using a well known graph algorithm, the depth-first search algorithm. This algorithm uses a stack were it keeps the nodes to be expanded and at each step it takes the element on the top of the stack, it generates the successors of the element and pushes them on the top of the stack. This algorithm explores all the paths from the nodes pushed on the stack at the beginning of the search and finishes when the stack is empty.

As we want to check if each scene is reachable from the root scene for each of its roles and if from each scene the final scene is reachable for each of its roles the search is made in terms of roles. This should not be a surprise as the paths that agents can follow within an institution are determined by the role they are playing. Concretely, the algorithm will explore the paths that can be followed with each role from the initial scene. In order to generate the succesors of a scene, each branch of the search always keeps the current role which is the role labelling the arc that lead to the last scene of the branch. The successors of a scene s_i in a branch of the search are the scenes s_j reachable from s_i with the current role. We have to take into account that as agents are allowed to change their role when traversing a transition the current role of a branch can change. That is to say, from a scene s_i with role r_k, it may be possible to reach a scene s_j with role r_q. Furthermore, we want to keep track of which scenes have been visited and with which role in each branch of the search. Then, for each branch of the search is maintained a sequence of pairs of scene and role identifiers. That is, each branch b_i maintains a sequence of the form $< s_0, r_0 > \ldots < s_n, r_n >$ where

$s_0 \dots s_n$ are the scenes ordered as visited in the branch and $r_0 \dots r_n$ represent the role when each scene was visited. Then, s_n represents the last scene of the branch and r_n the current role. This information is used to detect cycles and when the branch reaches the final scene for marking from which scenes and with which roles the final scene is reachable.

During the search, for each scene s_i two role lists are maintained, called the *visit* and *exit* lists. The first one, the *visit* list, contains the roles that can reach the scene s_i from the initial scene. That is, if a role r_j belongs to the *visit* list of scene s_i, it means that there is a path from the initial scene that permits agents to reach scene s_i playing role r_j. The second one, the *exit* list, contains the roles that can reach the final scene from the scene s_i. That is, if a role r_j belongs to the *exit* list of scene s_i, it means that there is a path from scene s_i to the final scene that agents playing role r_j in the scene can follow in order to leave the institution.

The search starts from the initial scene creating a branch for each of the roles that can be played within it. Then, as a first step of the algorithm for each role r_i that can be played in the initial scene s_0, it is pushed onto the stack the pair $< s_0, r_i >$. After that at each step of the algorithm it is obtained the element on the top of the stack which will be a sequence $< s_0, r_0 > \dots < s_i, r_i >$ representing a branch of the search. Then, it is checked if the scene s_i was previously visited with role r_i.

If it is the first time that a scene s_i is visited with role r_i, it is marked as visited for that role. That is, r_i is added to the *visit* list of scene s_i, denoting that scene s_i is reachable from the initial scene with role r_i. Then, the successors of scene s_i with role r_i are generated. As we have mentioned, these are the scenes reachable from s_i with role r_i traversing a transition. Then, for each scene s_j reachable with role r_j, which can be different of r_i, from s_i with role r_i the sequence $< s_0, r_0 > \dots < s_i, r_i >< s_j, r_j >$ is pushed on the top of the stack. If there are no succesors from the last scene with the current role the search in this branch finishes. That is, from s_i with role r_i no scene can be reached. There are two situations that can provoke that situation: that there are no outgoing arcs from s_i which contain r_i in its label, or that there are outgoing arcs from s_i to transitions that contain r_i in their label but there are no outgoing arcs from the transition that agents reaching the transition with role r_i can follow. In the first case, we have to distinguish if the scene is the final scene or not. If scene s_i is the final scene the graph is correct and the *exit* lists of the different scenes are modified as we explian later. Otherwise, an error has been found as we have found a scene which does not have any outgoing arc labelled by role r_i. In the second case it is always an error as we have detected some transition(s) where a variable appearing in its incoming arcs does not appear in any of its outgoing arcs.

The *exit* list is modified when a branch reaches the final scene. That is, when the final scene is a succesor of the last scene and role of a branch. When the final scene is reached it means that from all the scenes of that branch the final scene is reachable with the corresponding role. That is, if the sequence of the

branch is $< s_0, r_0 > \ldots < s_i, r_i >$, and the final scene s_f is a succesor of scene $< s_i, r_i >$, that means that for each pair s_j, r_j of the sequence, the final scene is reachable from scene s_j with role r_j. Then, for each pair $< s_j, r_j > \in < s_0, r_0 > \ldots < s_i, r_i >$, r_j is added to the *exit* list of the scene s_j.

What happens when a branch of the search goes to a scene with a role that was previously visited?. In this case we have to check if there is a cycle or not. That is, if the scene was previously visited in the same branch or in another branch of the searc. The performative structure is defined as a directed graph and it may conatain cycles. We have to take this into account for not entering in an infinite loop and in order to detect from which scenes and with which roles the final scene is reachable. A cycle is detected when the branch goes back to a visited scene in the branch and with the same role. That is, the next scene to visit is s_i with role r_i, the sequence of the branch is $< s_0, r_0 > \ldots < s_n, r_n >$, and $< s_i, r_i > \in < s_0, r_0 > \ldots < s_n, r_n >$. Notice, that if we go back to a visited scene but with a different role this is not a cycle. Of course when we detect a cycle we stop the search for that branch as we do not want to enter in an infinite loop and to explore again previously explored paths. For the first property that we want to check, we could forget about the cycle beacause we have marked as visited all the scenes forming the cycle with the corresponding role and all the paths from each scene and role of the cycle either have been explored or will be explored by the algorithm. But we want to know if the final scene is reachable from the scenes composing the cycle. The final scene is reachable from the scenes composing the cycle with their corresponding roles, if it is reachable from one of the scenes of the cycle with its corresponding role. That is, if there is a path from a scene of the cycle to the final scene whith the same role which has been visited within the cycle, there is a path from any scene of the cycle to the final scene with its corresponding role.

That is, if there is a cycle $< s_i, r_i >< s_{i+1}, r_{i+1} > \ldots < s_{i+t}, r_{i+t} >< s_i, r_i >$, and exists $< s_{i+k}, r_{i+k} > \in < s_i, r_i >< s_{i+1}, r_{i+1} > \ldots < s_{i+t}, r_{i+t} >$ from which there is a path $< s_{i+k}, r_{i+k} > \ldots < s_f, r_n >$ where s_f stands for the final scene, then for each $< s_{i+j}, r_{i+j} > \in < s_i, r_i >< s_{i+1}, r_{i+1} \ldots < s_{i+t}, r_{i+t} >$ exists a path $< s_{i+j}, r_{i+j} > \ldots < s_{i+k} r_{i+k} > \ldots < s_f, r_n >$ that goes from scene s_{i+j} with role r_{i+j} to the final scene.

Summarising, when a cycle is detected it is checked if the final scene is reachable from any scene of the cycle with the corresponding role. That is, for each pair s_j, r_j of the cycle, it is checked if role r_j belongs to the list *exit* of scene s_j. If this is the case, the final scene is reachable for all the scenes of the cycle with their corresponding role. Then, for each scene belonging to the cycle the role when it was visited is added to its list *exit*. Otherwise, the cycle is kept, that is, it is kept the list of scene and role pairs composing the cycle. At this point of the search, we can not be sure that all the paths from all the scenes of the cycle have been explored and then, a path to the final scene from a scene and role of the cycle can stil be found. Then, if during the search it is found a path from one of the scenes of the cycle with the corresponding role to the final scene, it means that the final scene is reachable from all the nodes of the

cycle. Otherwise, it means that the final scene is not reachable for the scenes composing the cycle with the corresponding roles.

The second case that we have to take into account is what happens when the search goes back to a previously visited scene with the same role but there is no cycle. That is, the search reaches a scene with a role visited in another branch. Then, we check if the final scene is reachable from the scene with the current role. If the current role belongs to the list *exit* of the scene it means that the final scene is reachable from this scene with this role. This means, that the final scene is reachable from all the scenes and roles of the current branch.

If the current branch is $< s_0, r_0 > \ldots < s_i, r_i >$, and the role r_i belongs to the list *visit* of scene s_i, and r_i belongs to the list *exit* of s_i, then, exist a path $< s_i, r_i > \ldots < s_f, r_n >$. Then, for each $< s_j, r_j > \in < s_0, r_0 > \ldots < s_i, r_i >$ exist a path $< s_j, r_j > \ldots < s_i, r_i > \ldots < s_f, r_n >$ to the final scene.

Then, for each pair s_j, r_j of the branch sequence we add the role r_j to the *exit* list of the scene s_j. What happens if the role does not belong to the list *exit* of the scene? As we are exploring the graph by a depth-first search, when the search goes back to a node that was previously visited in another branch, all the paths from the node have been explored. That means, that if there would be a path from that scene to the final scene for that role, it would have been found before and then the role would be in the *exit* list of the scene. Thus, if the role does not belong to the list *exit* of the scene, it means that there is no path from that scene with that role to the final scene and it makes no sense to continue the search for this branch.

If the performative structure is correct, at the end of the search for each scene its *visit* and *exit* lists must contain all the roles of the scene. This would mean that all the scenes are reachable for each one of the roles of the scene from the initial scene, and that from each scene and for each of its roles there is a path to the final scene that will allow agents to leave the institution. Notice, that the second property is not checked for those cases in which a scene is not accessible with a role from the initial scene. We have opted for this lazy solution in order not to start multiple searches. That is, to carry out one search for each role that can not reach a scene. As agents with that role will not be able to reach the scene if the institution is executed it does not matter if there is no path from this scene to the output scene for agents playing that role. For this cases the user is informed that the scene is not accessible from the initial scene with a concrete role. Then, when it will solve this error and the scene will be accessible with the role, it will be checked if there is a path with this role to the final scene.

Complementarily to the search algorithm, it is checked that for each scene all the roles appearing in its incomming and outgoing arcs belong to its role set and that each role appears at least in one incoming and one outgoing arc, that each scene has at least one incoming arc of type *new*, and for each transition it is checked that the set of variables appearing in its incomming arcs are the equal to the set of variables appearing in its outgoing arcs. In the case of transitions they must at least have one incomming and outgoing arc. That is, it is checked

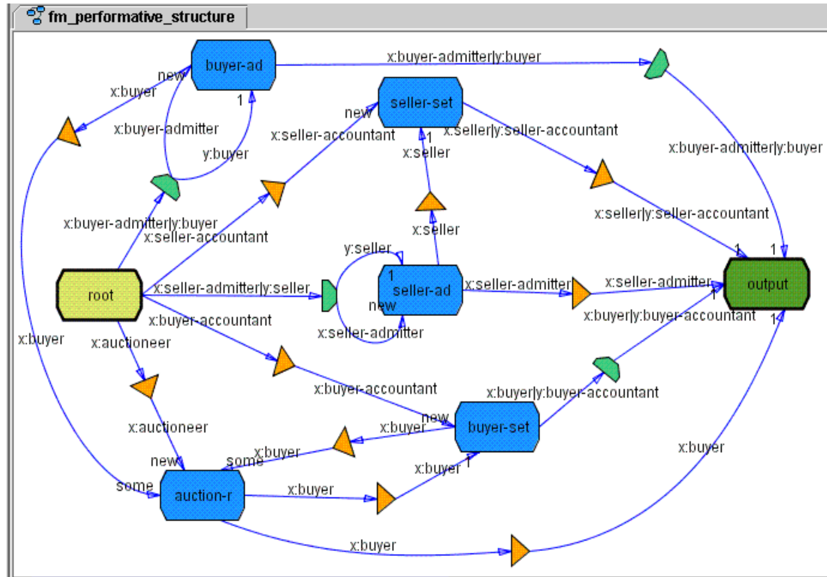


Figure 5.4: Fish Market Performative Structure

that it does not exist a disconnected transition.

As an example of how this works, we can see an example of how the algorithm works in a performative structure. Figure 5.4 depicts the performative structure of the Fish Market institution [Rodríguez-Aguilar et al., 1997]. The fish market is a trading institution devoted to the traiding of fish. In this figure we can see the different scenes and how agents depending on their role can move among them. The main scene is the auction room where products are auctioned but the institution also contains scenes for the admission of buyers and sellers, for accountability and the initial scene which is the scene *root*, and the final scene which is the scene *output*. In the institution there are two external roles which are the buyer and seller roles, and five internal roles in charge of the different scenes which are: auctioneer, buyer-admitter, seller-admitter, buyer-accountant, seller-accountant and auctioneer.

In figure 5.5 we can see how the search algorithm visits the different scenes of the fish market. Each node is labelled with the name of a scene and a role identifier, meaning that a role identifier, meaning that role. Each tree in the figure correspons to the exploration of the paths that can be followed by each of the roles from the initial scene. This result should not be a surprise, as the first step of the algorithm is to push on the stack for each role that can be played in the initial scene, the pair composed by the initial scene and the role. Each line on the tree corresponds to a succesor in the search, that is, if a node labeled by

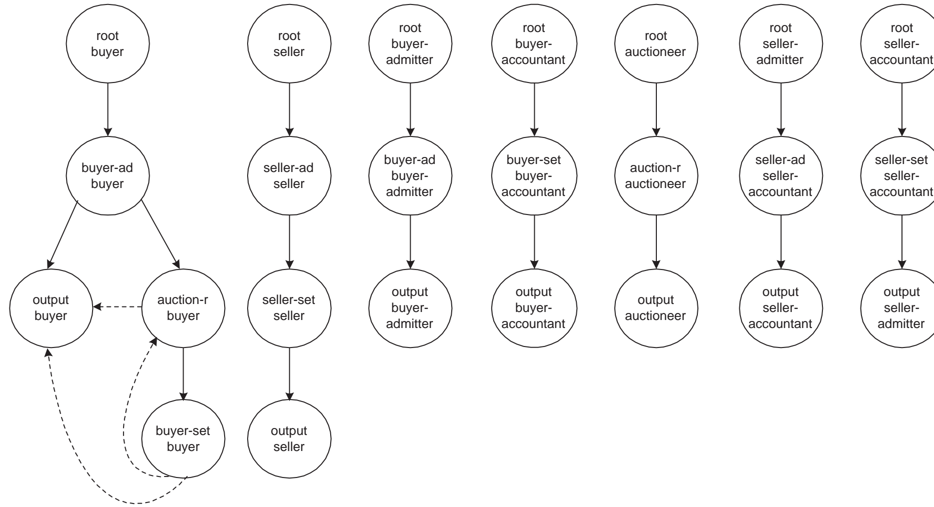


Figure 5.5: Depth-first search of the Fish Market performative structure

s_i r_i has a connection to a node s_j r_j, it means that agents playing role r_i in scene s_i can reach scene s_j with role r_j only traversing a transition. Dash lines correspond to the cases where the search goes to a previously visited scene and role. All the scenes are visited for each one of its roles and we can see that the final scene is reachable from all of the nodes.

We also have to guarantee that agents can entry and leave the scenes. In order to analyse the scenes we have to take into account that each scene specifies for each role a minimum and maximum number of agents that can participate playing that role and a set of access and exit states. Thus, the set of access and exit states determines the entry and exit points of the scene for a given role. When an agent wants to join a scene it has to wait until the scene reaches an access state for its role. The same applies for the exit states when it wants to leave the scene. The tool verifies that for each scene role there is at least one access state that will allow agents playing that role to enter the scene. Furthermore, the initial state must be an access state for those roles whose minimum is greater than zero, in order to start the scene, and for those roles labelling an incoming arc of type *new*, as agents following the arc must be incorporated to the scene execution once created. In order to allow agents to leave when the scene is finished, the final states must belong to the set of exit states for each one of the roles. Furthermore, arcs can be labelded with conjunctions of pairs of an agent variable and role identefier with the meaning that agents playing those roles have to progress together through the arc. Thus, if a conjunction appears in a scene incoming arc it is verified that exists an state which is an access state for all the roles appearing in the conjunction, while if the conjunction appears in an outgoing arc it is verified that exists a state which is an exit state for all

of the roles. Finally, it must be checked that every path that the conversation can follow from the initial state through the conversation graph leads to a final state. We will explain how this is checked in the next subsection when analysing the protocol correctness.

5.3.3 Protocol correctness

Another important verification step in the context of a scene is to check that the conversation protocol is correct. We demand a scene protocol to satisfy the following requirements:

- all the sates of the graph are accessible from the initial state.
- a final state is reachable from any scene state.
- the initial state is not reachable once left.
- there are no outgoing arcs from the final states.
- the illocution schemas labelling the arcs are correct with respect to the scene dialogic framework and ontology, and at least the terms referring to agents and time are variables.
- there are no outgoing arcs from the same state labelled with equivalent illocution schemes.
- all the occurrences of a variable within the scene have the same type.
- an application occurrence of a variable must be always preceded by a binding occurrence.

The first and second properties enforce that all the states of the conversation graph are accessible from the initial state and that from each of them there is a path to a final state. That is, that the conversation will not enter an infinite loop from which it can not go out and that there are no states, apart from the final states, from which there are no outgoing arcs. The combination of these two properties forces that for each state of the scene there must exist a path from the initial state to a final state which passes through the state. Moreover, the graph must satisfy that the initial state is not reachable once left, that is, the initial state does not have incomming arcs, and there are no outgoing arcs from the final states as they represent the different endings of the conversation.

Labels of the arcs must also be verified. As we want the scene protocol to be independent from concrete agents and time instants, we demand that in the illocution schemas that label the scene arcs, at least the terms referring to agents and time must be variables, while the other terms can be variables or constants.

In the definition of illocution schemas nothing is said about the types of the variables appearing in them, they have to be deduced using the position in which they appear in the illocution schemas and the ontology definition. Variables representing the sender and the receiver are associated to the type $AgentId$. If the

terms refering to the sender and receiver roles correspond to a role identifier this is also kept and it must be the same for all of the occurrences of the agent variable. That is to say, the same variable can not be used to represent an agent variable of different roles. If the terms referring to the sender and receiver roles are variables, then they are associated to the type $RoleId$. This is a special type only used in the verification. Otherwise, it is checked that they correspond to scene roles. The content of the illocution scheme is checked against the scene ontology. Remember that the content of the messages must be constructed making use of function definitions on the ontology returning a boolean and it must be expressed in the content language defined in the scene dialogic framework. Then, it is searched in the ontology for the function definition and if found, it is checked that the content expression is correct with respect of the definition and the types for the variables are deduced.

As we have said, scene variables must have the same type in all of its occurrences. As illocution schemas are analysed a table of symbols is created that keeps the type of each found variable. Then, when a new illocution scheme is analysed two situations can arise for each variable in the illocution scheme. On the one hand, if it is the first occurrence of the variable found the type deduced in the analysis of the new illocution scheme is assigned as the variable type within the scene. That is, the table of symbols is incremented by the new variable and its type. If an occurrence of the variable was found before, it is checked that the type deduced in the analysis of the new illocution scheme is the same than the one found before. If it is not, an error is given as the variable is being used within the scene with different types.

Also, the constraints and time-outs are analysed. In both cases they are defined as a list of expressions. The difference is that constraints expressions must be of boolean type while time-out epressions must be of a numeric type. For each variable appearing in the expression its type is looked up in the table of symbols. Then, taking into account the types of the variables, constants and the operators definition, it is analysed if the expression and sub-expressions are correctly typed. For each of them it is checked that the type of the arguments is correct with respect to the operator definition, and the type of the expression is the type returned by the operator with the types of the argument.

Remember that we distinguish between binding occurrences and application occurrences of variables. Since at the beginning of a scene all the variables are unbound, each application occurrence of a variable must be preceded by a binding occurrence. What does this condition imply? Imagine that in the label of an arc from state w_i to w_j appears an application occurrence of variable x, that is $!x$. Then, for each path from the initial state to w_i it must appear a binding occurrence of variable x, that is $?x$. Remember that binding occurrences can only appear in arcs labelled with illocution schemas, while application occurrences can appear in arcs labelled with illocution schemas and in arcs labelled with timeouts.

In order to check that each scene state is reachable from the initial state; that the final state is reachable from any scene state; and that each application

occurrence of a variable is preceded by a binding ocurrence, we explore the conversation using a depth-first search algorithm starting at the initial state, in a similar way as it is done at performative structure level. That is, the algorithm starts by pushing the initial state of the scene in the stack. During the search two global lists are maintained, one with the visited states, that is, the states which are accessible from the initial state, and another with the states from which the final state is reachable. We also maintain for each branch of the search a sequence of the states visited within the branch and the set of variables for which a binding occurrence has been found in the labels of the followed arcs. The sequence of states is used to detect cycles within a branch and for marking from which states a final state is reachable when a final states is found. When a branch reaches a final state, all the states of the sequence are added to the second global list, the one corresponding to the states from which a final state is reachable.

An important point to take into account is what happens when the search goes back to a previously visited state. That is to say, the search goes to a state which is marked as visited. There are two possibilities: that the state was visited in the current branch and then, there is a cycle; or that the state was visited in another branch. In the first case, we proceed in a similar way as in the performative structure, the search for this branch is stopped as we do not want to enter in an infinite loop, However, we have to check if a final state is reachable from the states composing the cycle. A final state is reachable from the states of the cycle if it is reachable from any of them. Then, it is checked if a state of the cycle which belongs to the list of states form which a final state is reachable exists. If this is the case, it means that from all the states of the cycle a final state is reachable, then all the states of the cycle are added to the list. Otherwise, the cycle is kept until the end of the search because a path from one of the cycle states to a final state can still be found.

When a state is visited again but we are not in a cycle we continue the search as if it was the first time that the state is visited. This is necessary in order to check that each application occurrence of a variable is preceded by a binding occurrence, as the set of variables for which a bindings ocurrennces has been found in the new path to the state can be different from the set of variables found in previous one(s). Hence, we have to continue exploring the graph in order to ensure that each application occurrence is preceded by a binding occurrence.

If the scene protocol is correct at the end of the search all the states of the scene must belong to the two global lists. That is to say, all of them have been visited and from of all them a path to a final state has been found. Furthermore, if the scene is correct all consulting occurrences of a variable must be preceded by a binding occurrence. Notice that for those states which are not accessible from the initial state it is not checked if a final stare is reachable from them. As this states are not reached, the paths from those states are not explored. Moreover, it is neither checked, if in the paths from those states each application occurences of a variable is preceded by a binding occurrence.

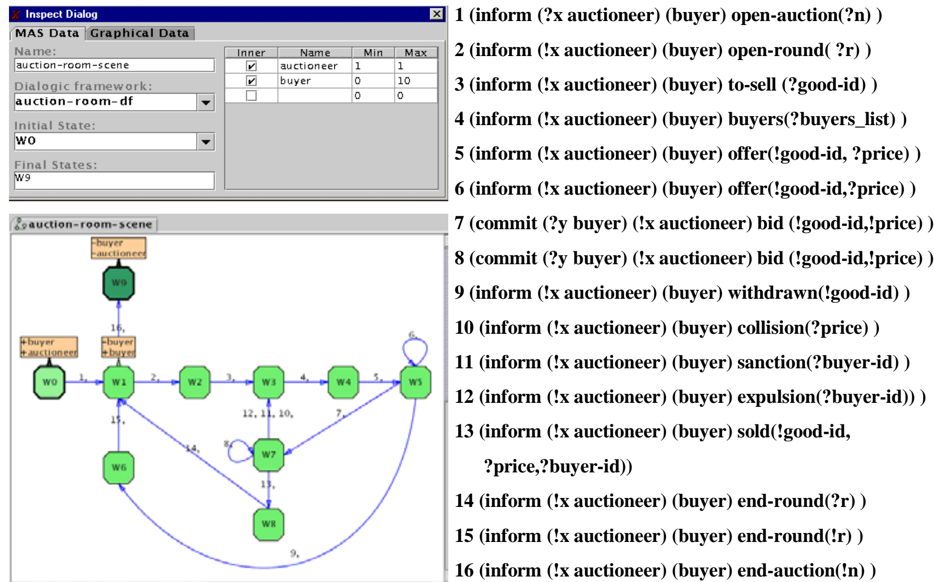


Figure 5.6: Auction scene following a Dutch protocol

In figure 5.6 we present the protocol for a scene where goods are auctioned following a dutch protocol. From the specification we can see that there are two roles that can be played within the scene, which are the auctioneer and buyer roles. We can also see that for each role there is at least one access state and that the final state is an exit state for all of them.

In figure 5.7 we can see how the search algorithm visits the diffrent states of the auction scene starting from its initial scene ω_0. The first branch finishes when the final state ω_9 is reached. The other branches finish when the search goes back to a visited state which is denoted by the dash lines. In all the cases there is a cycle as the search goes back to a state which was previously visited in the same branch. We can see that all the states are visited at least once, then all of them are accessible from the initial state. Also the final state is reachable from all of the states. On the contrary, w_0 and w_1 belong to the branch that reaches the final state, then the final state is reachable from them. The rest of the states belong to a cycle which contains w_1 and then, the final state is also reachable from all of them.

In table 5.1 we can see the types deduced for the scene variables when analysing the labels of the arcs, taking into account the following definitions

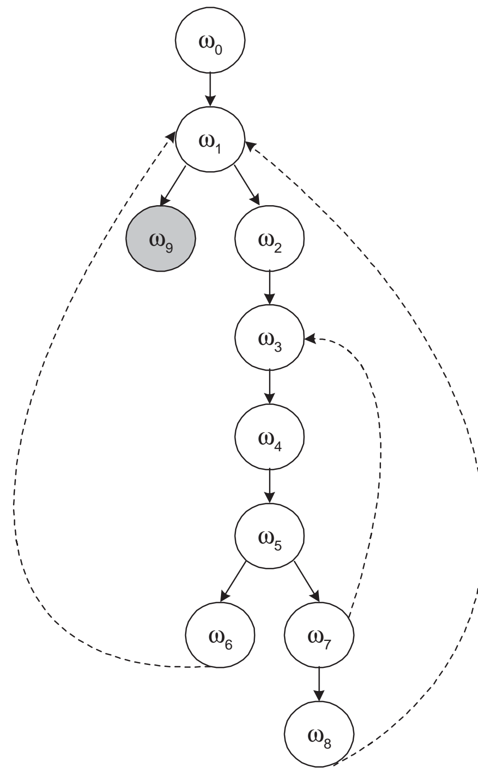


Figure 5.7: Depth-first search for the acution scene

from the scene ontology:

```
open-auction: numeric -> boolean
end-auction: numeric -> boolean
open-round: numeric -> boolean
end-round: numeric -> boolean
to-sell; string -> boolean
buyers: AgentId list -> boolean
offer: string * numeric ->boolean
bid: string * numeric -> boolean
withdrawn: string -> boolean
collision: numeric -> boolean
sanction: AgentId -> boolean
expulsion: AgentId -> boolean
sold: string * numeric * AgentId -> boolean
```

Variable	Type	Variable	Type
x	AgentId	buyers-list	AgentId list
n	numeric	price	numeric
r	numeric	buyer-id	AgentId
good-id	string	y	AgentId

Table 5.1: Deduced types for the variables of the auction scene.

5.3.4 Norm correctness.

Finally, the tool has to perform an important verification step which consists on checking that norms are correctly specified and that agents may fulfil them. Each norm specifies the actions that will trigger it, and the actions that agents must do in order to fulfil the obligations. As we have said, actions in a norm are specified as a pair formed by an illocution scheme and a scene. Then, for each pair scene variable appearing in the norm definition it must be verified that the scene exists and that an illocution matching the illocution scheme can be uttered within the scene.

As in the context of the scene, the type of the variables appearing in a norm definition must be the same for all of its occurrences. Then, each illocution scheme is evaluated in the context of its scene in order to deduce the types of the variables appearing in it. Another important restriction is that those agents for which an obligation is imposed must appear in all the illocution schemas in the antecedent and in the defeasible antecedent. This means that the illocutions which can impose or fulfil an agent obligations must be illocutions uttered or received by the agent.

Thus, we need to check that agents can fulfil the norms. That implies, that there must exist a path from the scenes where norms are activated to the scenes where they are fulfilled. There are no restrictions concerning the order in which the illocutions in the antecedent must be uttered and in which the illocutions in the defeasible antecedent must be uttered. This implies that the norm can be activated in any of the scenes appearing in the antecedent Thus, it must be checked that each scene appearing in the defeasible antecedent is accesible from each scene appearing in the antecedent. Therefore, for each scene in the antecedent a depth-first search with the role of the agent that will acquire the obligations is made. The goal of the search from each scene of the antecedent is to check that all the scenes appearing in the defeasible antecedent are accessible from it. Then, the search stops when all the scenes appearing in the defeasible antecedent have been visited with the correspondig role.

5.3.5 Model Checking Scenes

One of the advantages of having the specifications in a textual format is that it they can be easily translated into preexisting languages and benefit from the work done in those languages.

This section, which is a brief summary of [Huguet et al., 2002][2], we report the work done exploring how model checking can be used to verify electronic institution. Concretely, model checking ir is used to verify some properties of scenes.

For this reason scene specifications are translated from its ISLANDER textual representation into MABLE a language defined for the design and development of multiagent systems [Wooldridge et al., 2002]. MABLE language is an imperative language similar to C and each agent defined in MABLE has a mental state consisting of "beliefs", "desires" and "intentions". The MABLE language has been implemented making use of SPIN [Holzmann, 1997], an available model checking system for finite state machines. Agent definitions in MABLE may incorporate claims about its behaviour which are expressed in a quantified multi-modal temporal logic and can be automatically checked.

As scenes are specified in ISLANDER as a type of finite state machines they can be easily translated into MABLE. The claims that we want to prove are expressed in quantified temporal logic. The code in MABLE is translated into promela, the system description language used by the spin model checker. The claims that can be currently checked correspond to the checking that from each scene state a final scene state is reachable. Concretely, it is checked the oposite claim, which is, whether it is impossible to reach the final state from a given state. This is expressed as follows:

$$\Box[(state = W_i)] - > \Box \neg exit]$$

For each state of the scene a claim is so generated and it is checked whether these claims are false. If a claim violated message is generated for each claim, that means that the final scene is reachable from each state. The results of this work show how specifications in ISLANDER can be translated into MABLE and how claims on the generated translation can be defined and proved. This opens the opportunity to look for more interesting properties to be checked.

5.4 Conclusions

We think that a formal specification is needed before starting the development of complex systems. This is also true for Multi-agent systems. This formal specification process allows to identify the important parts of the system and detect possible errors saving time for the designer. Furthermore, the existence of software tools that help the designers on the specification and development of multi-agent systems is a key aspect for the succes of the area. This is specially true for the expansion of use of multi-agent systems in software companies.

In this line we have presented ISLANDER a tool for the specification and verification of electronic institutions. First, we have defined a textual language to

[2]This work has been done in colaboration with Marc-Philipe Huguet, Steve Phelps and Michael Wooldridge of the Liverpool University. They have also implemented the software tool which translate ISLANDER specifications into MABLE and verifies them.

specify institutions based on their formalisation presented in chapter 3, and the ISLANDER editor that permits the graphical specification of several language components. We think that this facilitates a lot the work of the designer because graphical specifications are extremely easy to understand and they are similar to the informal diagrams that engineers use while designing, constructing and analysing a system. Graphical specifications can also be used as a presentation of the specified institutions.

Once the specification is finished it goes through a validation process. As several components of an institution are specified as a graph, an important part of the verification is done using graph algorithms. It is verified that agents can reach the different scenes in which they can participate, that agents have always a path that will allow them to leave the institution that each conversation is specified correctly and that agents can fulfil the obligations that they acquire as a consequence of their actions within the institution. In this verifications performative structure's and scene's constraints are not taken into account. Hence, the next step will be to take them into account when verifying liveness and protocol correctness. As specifications generated by the ISLANDER editor are used in different stages on the development of infrastructures and agents for the specified institutions it is crucial to ensure that specifications do not contain any error.

Chapter 6

Social Layer for Electronic Institutions

In previous chapters we have presented our electronic institution model and how electronic institutions can be specified and verified. That is to say, which are the components of electronic institutions and the properties that these components must satisfy. In this chapter we focus on the execution of institutions and we present a generic infrastructure for them that we have developed. Thus, the developed infrastructure can be used in the deployment of different institutions. The purpose of the infrastructure is to facilitate agent participation and communication in the institution, and to enforce the institutional rules to them.

Participants in electronic institutions are heterogeneous (human and software) agents (possibly) written by different people, in different languages and with different architectures. No assumptions are made about the characteristics of the participating agents, nor restrictions are imposed to its designers and in which environment the agents run. We believe that such assumptions have some advantages as agent designers can choose the language and architecture that is better to fulfil their goals, facilitate the reusability of previously developed agents, and the integration of existing software and AI components. On the contrary, we can not assume that these agents will behave according to the institutional rules. Then, the institution infrastructure is in charge of guaranteeing that agents actions within the infrastructure do not violate the institutional rules.

Instead of developing an infrastructure from scratch, we have opted for developing a social layer middleware on top of a communication layer implemented by the JADE platform. The social layer middleware is thought to guarantee that agent interactions are structured according to the norms and conventions defined in the institution specification. Next we summarise the main functionalities of the social layer middleware:

- facilitate participating agents the information they need to successfully participate in the institution;

- facilitates their communication with other agents within the different conversations;
- guarantees the correct evolution of each conversation;
- prevents errors made by the participating agents by filtering erroneous illocutions, thus protecting the institution; and
- controls which obligations participating agents acquire and fulfil.

Another important aspect is the development of agents which can participate in the institution. Specially in the case of staff agents, those that play internal roles, as the institution delegates its services and duties to them. The specification defines *what* agents can do within the institution but not how they have to take their decisions. Hence, agents can not be completely generated from the specification. Instead, skeletons of agents can be obtained from the specification which agent developers can customise in order to completely develop their agents. In the last part of the chapter we explain how agent skeletons can be obtained from ISLANDER specifications and how skeletons can be customised by agent engineers.

The chapter is structured as follows. In the next section 6.1 we describe the institution architecture. In section 6.2 we present a brief description of JADE as it is the FIPA-compliant platform that we have chosen as our communication layer. Next, in section 6.3 we focus on the social layer with a special attention on the governors because they are the more complex and important of the agents composing the social layer. In sections 6.3.2, 6.3.3 and 6.3.4 we concentrate on how scenes, transitions, and norms, are handled by the agents of the social layer. Next, in section 6.4 we explain how agents for an institution can be developed using institution specifications. Finally, in section 6.5 we summarise the chapter.

6.1 Institution architecture

Figure 6.1 depicts the architecture of an electronic institution. We have opted for a multi-layered architecture composed by the following layers:

- autonomous agent layer: composed by the agents taking part in the institution.
- social layer: devoted to facilitate agent participation within the institution and to enforce the institutional rules encoded in the specification.
- communication layer: it is in charge of providing a reliable and orderly transport service to the agents composing the social layer.

Notice that participating agents in the institution do not interact directly, they have they interactions mediated by the social layer. In order to know the valid interactions that agents must have, and the rest of the characteristics of

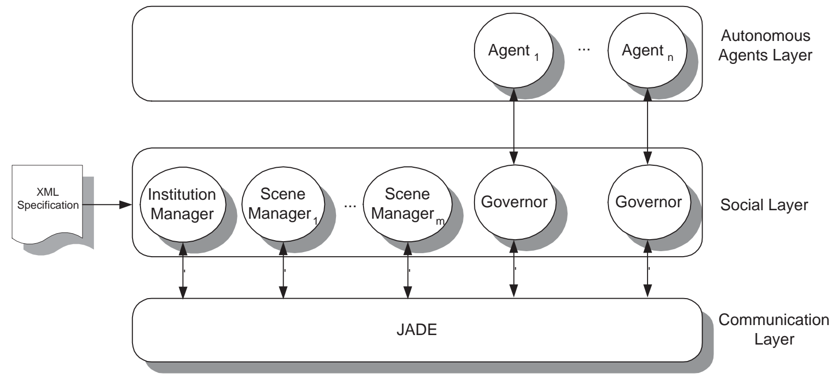


Figure 6.1: Institution architecture

the institution the agents of the social layer load the institution specification from its XML representation as generated by the ISLANDER specification tool.

Building the social layer on top of JADE eases the communication of the agents of the social layer as they do not need to deal with low level communication issues. For instance, agents do not need to know the physical addresses of other agents as the JADE transport service is in charge of routing messages and deliver them to their addressees. Furthermore, the monitoring and debugging tools offered by JADE have been very useful for testing the agents of the social layer. All the agents of the social layer have been developed in JAVA as JADE agents.

In the following sections we will present an overview of JADE, we will describe in detail the social layer, and we will explain how agents for an institution can be developed.

Before detailing our infrastructure we want to compare it with the infrastructure proposed in [Rodríguez-Aguilar, 2001] since our work is a continuation of it. Rodríguez proposes that the infrastructure can be realized making use of a special type of mediator agents, the so called *interagents* [Martín et al., 2000] devoted to the mediation of agent interactions. Each agent in a conversation is connected to an interagent which mediates the interaction between the agent and the society where it is situated. Interagents of the different agents taking part in a conversation coordinate in order to guarantee the correct evolution of the interaction. A main feature of interagents is that the conversation protocol that they manage is the agent view of the protocol, and not the protocol as a whole. That is, the conversation protocols that they manage only include messages where its associated agent is involved and arcs' labels include information on whether the transition will be provoked by either the transmission or the reception of a message. This can be seen as the role projection of the global protocol. Hence, interagents in the same conversation manage complementary conversation protocols. That is to say, when one interagent is in a state where a transition can be provoked by the emission of a message, the interagent of the

agent to which the message is addressed must be in a state where a transition can be provoked by the reception of the message. Among the interagents of the agents taking part in the same conversation one plays the role of the *leader* and the rest play the *follower* role. It is a duty of the leader to decide which illocution is accepted when more than one agent can speak. In this case, all the illocutions are sent to the leader which decides which of them is accepted and makes the scene evolve. The interagent playing the *leader* role is also in charge of authorising agents to join and leave the conversation. Interagents can manage conversations where only two agents are involved and conversations where multiple agents are involved but where interaction is always between one agent and the rest of the agents. This kind of conversations can be managed by an interagent as a set of one to one conversations. On the contrary, they fail to manage conversations where each agent can speak to all other agents. We have improved this limitation in this Ph.D.

Rodríguez's infrastructure also contains another type of agent, the so-called *institution manager*, which is in charge of authorising agents to join the institution and of managing the transitions. During the institution execution it keeps information about all the participants, and about all the current scene executions, for each one it keeps which are the participants within the scene and which is the interagent leader of the scene execution. Summarising, the institution manager is in charge of authorising agents to join the institution, their movements among performative structures' scenes, as well as the creation of new scene executions while management of scenes and control of agent pending obligations are kept as duties of interagents. These ideas were validated by the development of the fish market institution [Rodríguez-Aguilar et al., 1998].

Following Rodríguez proposal we have in our infrastructure an institution manager in charge of the same duties, and each participating agent is connected to a mediator agent, which we call *governor* [1]. The evolution of the institution formalisation, the limitations of the interagents in the management of complex interactions protocols, as scenes have become, and the goal of having an infrastructure capable of loading institution specifications as generated by the ISLANDER editor, has motivated us to develop a new infrastructure. Governors are in spirit similar to interagents and are in charge of similar duties, but there are some differences. Concretely our model differs from the one proposed in [Rodríguez-Aguilar, 2001] in the following issues:

- the conversation protocol managed within scenes;
- the scene management;
- the infrastructure architecture;

Firstly, governors manage global interaction protocols in contrast to the agent view of the protocol managed by the interagents. Furthermore, governors are capable of managing timeout transitions and constraints, being both extensions of scenes added after the development of interagents. Thus, governors are no

[1] We borrow this name from Noriega's thesis [Noriega, 1997]

limited on the type of scenes that they can manage. Secondly, in our approach we do not differentiate a *leader* among the governors of a scene. The duties of the *leader* were to decide which illocution makes the scene evolve when more than one agent could speak and to authorise agents to join and leave the scene. These duties are now undertaken in the following way:

- Governors implement a token-passing protocol to guarantee that only one agent speaks at the same time.
- The control over which agents join and leave the scene is devoted to a new type of agent, the so-called *scene manager*.

Then, for each scene execution there is a scene manager being responsible for authorising agents to join the scene and for participants to leave. We think that with the Rodríguez's approach the agent whose governor is the leader could be penalised with respect to the others, as its governor is in charge of more duties. Lastly, we have divided the infrastructure in two layers: a communication layer for which we use JADE and a social layer which contains the institution manager, the scene managers and the governors. In this way agents of the social layer do not need to take care of low level communication issues as JADE provides them a reliable transport service. In Rodríguez's approach the agents composing the infrastructure were in charge of guaranteeing a reliable transport service and they had to maintain information about the physical addresses of the other agents.

6.2 JADE in brief

In this section we offer a general overview of the Java Agent Development Environment (JADE), the FIPA-compliant platform that we have chosen as communication layer. For a more detailed description about JADE readers are referred to [JADE, URL, Bellifemine et al., 2001, Bellifemine et al., 2002b, Bellifemine et al., 2002a].

JADE is a software development framework which contains a FIPA compliant agent platform developed in JAVA and a package to develop JAVA agents. The main goal of JADE is to simplify and facilitate the development of multi-agent systems. Apart from the FIPA compliant platform which permits agents executions, it provides support to agents development and as well as different tools that permit to manage, monitor and debug the platform execution.

As a FIPA compliant platform, JADE contains the following mandatory roles performed by different agents:

- **Agent Management System (AMS)**. It controls the platform execution determining which agents can be executed, it maintains a list of the agents running in the platform and it handles their lifecycles.
- **Agent Communication Channel (ACC)**. Responsible for the agent communication inside and outside the platform. It guarantees a reliable, orderly, and accurate message routing service.

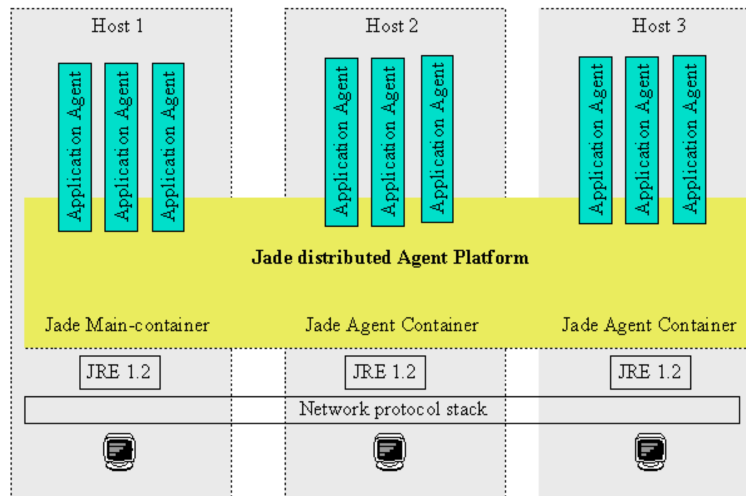


Figure 6.2: Architecture of the JADE platform

- **Directory Facilitator(DF)**. It provides a yellow pages service to the agents of the platform. Agents can register their services in the DF, and request the services offered by other agents.

The platform is perceived from outside as a single entity but it can be divided in different agent containers, each one executing a Java virtual machine and it can be distributed in different hosts. Figure 6.2 depicts the scheme of a JADE platform execution. When the platform is launched the AMS, the DF and the ACC are started. Each agent in JADE is executed within an agent container and it must have a global unique identifier within the platform which is composed by the agent name and the platform where it is running. Concretely, the global unique identifier of an agent is constructed, like e-mail addresses, by the name of the agent, a '@' symbol, and the address of the platform. Agent containers communicate using Java RMI while communication with other FIPA compliant platforms is done using IIOP. The main container is the one containing the AMS, the DF and the RMI registry. The other containers, once launched, connect to it.

JADE defines a generic agent class that agent developers must extend in order to program their agents. Developed agents inherit from its superclass different services such as: the capability of registering and deregistering in the platform, a set of basic methods for sending and receiving messages, the capability of cloning, etc.

An agent functionality is defined by a set of behaviours which represent

logical threads. So, each agent is composed of a set of behaviours which can be either added or removed dynamically. Behaviours represent logical threads but each agent in JADE has only one execution thread associated and behaviours are executed in a round-robin non preemptive. That is, behaviours are never interrupted and so they are executed until they return. Hence, it is important to define simple behaviours and be careful to avoid behaviours with an infinite loop (in this case since the behaviour never returns, no other agent behaviour will be executed). During its execution each agent maintains a queue of active behaviours, a queue of blocked behaviours and a queue of incoming messages. When a behaviour finishes, the next one in the queue of active behaviours is executed. When the agent receives a new message is added to its queue of incoming messages and all its blocked behaviours are activated by moving them to the queue of active behaviours. Messages in the queue can be accessed by blocking and unblocking operations and in both cases a message template can be passed. In this case, the returned message must match the given template. For blocking operations it can be defined a maximum time to wait for the message.

Agents within JADE exchange messages in FIPA ACL. Moreover, JADE has implemented some FIPA protocols and offers facilities for the definition of user ontologies and new content languages.

One of the main features of JADE is that it offers a set of tools for the management, monitoring and debugging of the platform. These tools allow the user to know what is happening within the platform and to interact with it. Due to the complexity of testing and debugging distributed systems these tools largely facilitate the developers work. More precisely JADE offers the following tools:

- Remote Monitoring Agent (RMA). It shows all the information about the platform, which agents are being executed in each container with their IDs, the state of each one, etc. Using the RMA the user can modify the execution of the platform. For instance, creating new agents, eliminating a participating agent, or shutting down a container. In order to obtain the information and to execute the required operations it uses the AMS agent.

- Directory Facilitator GUI. It permits users to see the agents and services registered in a Directory Facilitator agent. Furthermore, users can modify agents and services information.

- Dummy Agent. It is a graphic user interface which permits users to act as an agent in the platform. Using the dummy agent users can send messages to the other agents and are informed about received messages.

- Sniffer Agent. It permits to keep track of the messages exchanged by an agent or a group of agents. It shows all the messages sent and received by the selected agent(s) to the user. This permits the user to investigate the exchange of messages among the agents connected to the platform.

- Introspector Agent. This is the last tool developed, and it permits to follow

and control the life cycle of an active agent and to see the messages that it sends and receives.

6.3 Social layer

In figure 6.1 we differentiate three types of agents composing the social layer:

- **Institution Manager**. It is in charge of starting the system, authorising agents to enter the institution and controlling their movements between scenes. During the institution execution it maintains a list of all the agents taking part in the institution and a list of the active scenes along with the scene manager in charge of it. As it knows at every moment the current executions of each scene, it is in charge of authorising agent movements to current executions and the creation of new scene executions. Then, for each transition it keeps the agents within the transition and the scenes that they have requested to go to. There is one Institution Manager per institution execution.
- **Scene manager**. It is the responsible for governing a scene. It is in charge of authorising agents to join or leave the scene at the specified states whenever the restriction on the minimum and maximum agents per role is not violated. There is one scene manager per scene execution.
- **Governor**. Each governor is devoted to mediate the communication of an agent and the rest of the agents within the institution. They are in charge of giving participating agents all the information they require for participating in the institution. They also check that agent actions are correct with respect to the specification and the current execution. There is one governor per participating agent.

The social layer can be distributed among different machines for scalability purposes. Such distribution does not require modifying the agents as they communicate in the same way with any agent running on a JADE platform. Notice that the number of agents in the social layer change as new agents join and leave an institution, and as scene executions are created and finish. Conceptually, governors and scene managers are created by the institution manager which is in charge of authorising agents to join the institution and the creation of new scene executions, But there is a technical limitation because an agent does not have the capability of creating agents in other machines. To solve this problem an agent that works for the institution manager must be created in each machine to be responsible for creating governors and scene managers on behalf of the institution manager.

The social layer that we have developed is generic in the sense that can be deployed to realise different institutions. Agents composing the social layer load institution specifications as XML documents generated by the ISLANDER specification tool presented in chapter 5. Thus, changes on an institution specification solely involve the loading of a new XML document. From the institution

specification the agents of the social layer know the static information of the institution. That is, which are the roles that can be played within the institution, which are the different scenes and their protocols, how agents can move among scenes depending on their role and which are the institution norms. This information along with the information of the current execution, will be used to validate the agent actions within the institution.

The execution of an institution starts with the creation of an Institution Manager. Once up, the institution manager activates the initial and final scenes launching a scene manager for them. Thereafter, agents can start to request to join the institution upon requesting the institution manager. When an agent is authorised to join the institution, it is connected to a governor and initially admitted into the initial scene. From there, agents can move around the different scene executions or start new ones according to the specification of the institution performative structure.

The current version of the social layer presents some limitations in the institutions that they can manage. As explained in chapter 3, performative structure arcs are labelled by conjunctions and disjunctions of pairs composed of an agent variable and a role identifier, where a conjunction means that a group of agents must progress together through the arc. Furthermore, the labels on arcs connecting scenes and transitions can also contain conditions that agents must fulfil in order to progress through arcs. However, the current version of the social layer does not support these features. Concretely, agents can only progress together after an *And* transition and conditions are not taken into account when agents move from scenes to transitions. That is to say, conjunctions can only appear in outgoing arcs of transitions of *And* type and agents within scenes are allowed to move to any transition reachable from the scene with their role.

Next, we focus on the governor because it is the more complex and important of the agents composing the social layer. We explain the governor architecture, the communication channels that it may use to communicate with its associated agent and the messages that they can exchange. Later on, in sections 6.3.2, 6.3.3 and 6.3.4, we will focus on how scenes, transitions and norms are managed by the agents of the social layer.

6.3.1 Governor

Architecture

Each participating agent in an electronic institution is connected to a governor which mediates its communication with the rest of the agents. The communication between a governor and an agent is structured in conversations as shown in figure 6.3. There is always a conversation between the agent and its governor devoted to allow the agent to request general information about the institution execution and to inform it about the adoption and fulfilment of obligations. This conversation is created when the governor and the agent are connected, and it only finishes when the agent leaves the institution. The other conversations correspond to the scenes and transitions in which the agent is taking part. They are

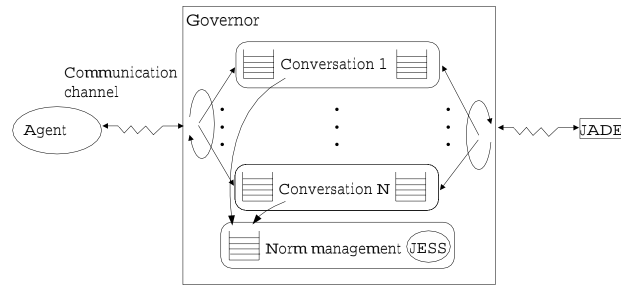


Figure 6.3: Governor architecture.

created and destroyed dynamically as a consequence of agent movements within the performative structure.

An important issue is which information should be provided to a participant about the institution execution. That is, information about the participants within the institution and about the different scene executions. Imagine, for instance, a trading institution where agreements among traders are reached by using a one to one negotiation scene. Probably, it would be not be desirable to inform a trading agent about the current negotiation scenes and the participants in each one, or to inform an agent about other agent's obligations. We believe that the decision about which information is provided to each agent should be decided when designing the institution and it should be associated to roles. In this way agents playing the very same role will have access to the same information. Hence ISLANDER should be extended to permit the definition of which information is provided to each role. In the current version of the social layer this conversation has been restricted to the communication of obligations information.

A governor can receive messages from the agent to which it is associated and by the other agents composing the social layer. Messages from the agent are received through its communicating channel while messages from the other agents of the social layer are received through JADE. In section 6.3.1 we detail which communication channels can be established between an agent and its governors. For each conversation the governor has two queues, one for the messages received from its associated agent and one for the messages received from the social layer agents.

The *Governor* has an execution thread that listens all the time to the communication channel with the agent in order not to lose any message. When a message is received the thread checks to which conversation the message is addressed to add it at the end of the corresponding conversation queue. Later on, the thread in charge of this conversation will take the message and process it. In order to avoid that the agent may overload its *Governor* with a massive sending of messages, the length of the reception queues for each conversation is limited to twenty messages.

On the other hand, messages received from the other agents of the social layer

are gathered by a behaviour that places them in another conversation queue. In this case, an execution thread is not used because JADE places the received messages in the agent's private queue, so that messages are not lost. Moreover, JADE will wake up the behaviour as new messages arrive to save CPU time. This queue is not limited by two reasons:

- the retransmission of these messages would require establishing complex control mechanisms that would reduce the system's performance.
- because these are messages received from the other agents of the social layer which only send to the governor the messages needed for their correct coordination.

The *Governor* has an independent execution thread to manage each conversation. We have discarded to use JADE behaviours because in that case agents holding in more conversations would be penalised because of the policy that JADE follows after the reception of a message. When a message arrives, JADE wakes up all blocked behaviours by moving all of them to the queue of active behaviours. Remember that JADE executes the active behaviours of an agent following a *round-robin non-preemptive* policy. Hence that in order to handle two successive messages for the very same conversation it would be necessary to execute the behaviours of all other conversations. Thus, agents which are taking part in more conversations would be penalized with respect to agents taking part in less conversations. In our approach conversation threads are kept asleep while there are no messages for the conversation they are handling, and they are woken up when a message addressed to their conversation arrives. That is, when a message arrives it is read by the governor behaviour or the thread in charge of the communication channel, and in both cases the message is added to the corresponding queue of the conversation to which the message is addressed and the thread handling the conversation is woken up to process the message.

Governors have another thread devoted to the management of norms. During the agent participation within the institution the governor keeps a list with the agent pending obligations. In order to chek whether an agent either acquires new obligations or it fulfils some of its pending obligations, we advocate for the use of the Java Expert System Shell (JESS). Then, this thread is in charge of adding the rules and facts to JESS, and it is informed whenever a norm is activated or some obligations are fulfilled. In both cases the agent is informed about the new obligations that it has acquired or about the obligations that it has fulfilled. For further details about norms management the reader is referred to section 6.3.4.

Agent-Governor communication channel

Nowadays there are many well established communication channels (USB, infrared, socket, etc) that carry information in different ways. In order to allow the agent to use any of these channels or any future one to communicate with its governor, we have built an abstract model of them. One of the advantages thus obtained is that it is not necessary to design a specific *Governor* for each

channel. Thanks to the use of java, that follows an object-oriented methodology, it has been possible to do it in a simple way with two classes: an abstract one called *Link*, and another one called *Message*.

The *Governor* uses the class *Link* to have access to the functionality of any channel in the same way. The *Link* class provides the methods to send and receive messages, to open and close a channel, and to check that the channel is not broken. In order to create a new communication channel it is only necessary to extend the class and implement the methods. At present we have implemented two communication channels: one via *socket* so that most of the languages and systems allow its use, and another one to receive messages from JADE. This allows participating agents to be in any FIPA compliant platform as all of them can be connected with JADE.

The *'Message'* class makes the information that travels over the channel independent from its codification. When a message has to be sent through a channel, the *'Link'* class, in charge of the management of the channel, extracts the information from an object of class message, and codifies it in a way that makes it possible to travel through the channel. The reverse operation is done on the receiver side, that is, the *'Link'* class at the receiver side decodifies the information received through the channel and it creates a new object of class *'Message'* with the received information.

Agent-Governor communication Protocol

One of the most important aspects of our implementation is the communication protocol between governors and participating agents. Since an agent can only communicate with its governor, any action an agent can do, any information that it can request and any information it can receive must materialise as a message to or from its governor. Apart from the result of the actions that an agent wants to do, about the information it requests, and the messages addressed to it, the governor sends the agent all the information it needs to successfully participate in the institution. In other words, the governor informs the agent about all events that occur within the institution that are relevant to the agent. For instance, the agent is informed when a new agent joins a scene in which it is taking part, or when a norm implying new obligations for it is triggered.

The agent and the governor exchange messages in FIPA-ACL where the content has the following elements:

- ConvID: an integer identifying which conversation the message belongs to.
- Action: the action to do or the action the receiver is informed about.
- Parameters: additional information needed to specify the action.

Agents intending to act within the institution or requesting information about the institution itself send request messages to its governor. Table 6.1 lists messages that an agent can send to its governor, except the first one for entering the institution which is sent to the institution manager. The table

	Action	Description
1	enterInstitution	Request to enter the institution
2	saySceneMessage	Request to say a message in a scene
3	moveToTransition	Request to move from a scene to a transition
4	moveToScenes	Request to move from a transition to several scenes
5	accesScenes	Ask for the scenes the agent can move into from a transition
6	accesTransitions	Ask for the transitions the agent can move into from a scene
7	agentObligations	Ask for the pending obligations
8	sceneState	Ask for the current scene state
9	scenePlayers	Ask for agents in a scene

Table 6.1: Messages that an agent can send to its governor.

contains a message per action or information request. As to actions within a institution, an agent can say a message within a scene in which it is taking part, move from a scene to a transition and move from a transition to a set of scenes. As to information request, an agent can ask its governor for the scenes it can reach from a transition, the transitions that it can reach from a scene, current agent's obligations, and the current participants or state of a scene in which it is taking part. An agent can cancel any sent message by sending a cancel message before the request has been processed.

When the governor receives a message from the agent the thread in charge of the communication channel processes it and then the governor answers the agent with an *agree*, *refuse* or *unknown* message. If the governor answers with an *agree*, it means that the message is sintactically correct and that it has been added to the queue of messages of the corresponding conversation. If the governor answers with a *refuse*, it means that the message can not be added to the queue of messages of the conversation because it is full or because the message is incorrect. A message can be incorrect because its *ConvId* refers to a non-existing conversation or because the message was found to be syntactically incorrect. Anyhow the agent is informed about the reason that made the message to be refused. Finally, an *unknown* message means that the governor did not understand the message.

Correct messages will be processed later on by the thread in charge of the corresponding conversation. After processing the message the governor will respond the agent with either the information requested or the result of the requested action. Table 6.2 shows all the messages that a governor may send to its associated agent. For any action that an agent can do within the institution there are two messages: one informing that the action has been done and another one informing that the governor failed trying to perform the action. In this later case, the message contains information about the reason that made the action

	Action	Description
1	enteredInstitution	The agent has entered the institution
2	exitedInstitution	The agent leaves the institution
3	enteredInstitutionFailed	The agent could not enter the institution
4	saidSceneMessage	An agent message has been said within a scene
5	saySceneMessageFailed	Agent message in a scene has failed
6	receivedSceneMessage	Reception of a message for the agent within a scene
7	timeoutTransition	The scene state has evolved as a consequence of the expiration of a timeout
8	enteredAgent	An agent has entered the scene
9	exitedAgent	An agent has left the scene
10	finishedScene	The scene has finished
11	currentAccesScenes	List of all the scenes that the agent can move into from a transition
12	movedToScene	The agent has entered a scene
13	moveToSceneFailed	Agent attempt to move to a scene failed
14	currentAccessTransitions	Informs of all the transitions that the agent can move into from a scene
15	movedToTransition	The agent has been moved to a transition
16	moveToTransitionFailed	Agent attempt to move to a transition failed
17	acquiredObligations	Informs of acquired obligations by the agent
18	obligationsFulfilled	Informs of fulfilled obligations by the agent
19	currentObligations	Informs about the current obligations of the agent
20	currentSceneState	Informs about the scene state
21	currentScenePlayers	Informs about the agents within a scene

Table 6.2: Messages that the governor can send to its associated agent.

fail. For instance, it is trying to utter an illocution which it is not correct with respect to the current execution of a scene.

The table also contains the messages that the governor sends to the agent in response to requested information. That is, messages informing the agent about: the scenes it can reach from a transition, the transitions that it can reach from a scene, the agent current obligations, and the current state or participants within a scene in which the agent participates. Furthermore, governors pass the agents the messages in the scenes executions addressed to them and inform them of the following events: agents entering or leaving a scene in which the agent is taking part, the end of a scene in which it is taking part, the adoption of new obligations, and the fulfilment of pending obligations. Notice, that if agents were allowed to ask for general information about the institution execution more messages must be supplied for requesting and receiving information.

6.3.2 Scene Management

In this section we focus on the execution of a scene. In the execution of a scene several agents in the social layer are involved, namely: a scene manager, and one governor per agent taking part in the scene. They must coordinate in order to guarantee its correct evolution. The execution of a scene starts with the creation of a scene manager endowed with the scene conversation protocol, the roles that participating agents may play and the maximum and minimum number of agents per role that can participate in the scene in order to regulate requests to join and leave. Once the scene manager is up, running agents may start to join the scene. Notice that the scene conversation protocol can not start until the minimum number of agents per role is reached.

A scene conversation protocol defines all possible interactions that agents may have defining at each moment (state) what can be said, by who and to whom. When a scene is played by a group of agents incarnating its roles, agents make it evolve by the utterance of grounded illocutions which match the illocution schemes labelling the scene arcs and that satisfy the constraints associated to them.

In order to evaluate agent actions within a scene, governors and scene managers use the scene specification and the contextual information of the current execution. For each scene execution they keep the following contextual information:

- all the participating agents with the role they are playing within the scene;
- the current state of the scene execution; and
- all the variable bindings made up by uttered illocutions(Σ), where Σ stands for the sequence of all the substitutions done during a conversation, i.e. a sequence of $\sigma_{w_i w_j}$ each one corresponding to an actually uttered illocution from state w_i to state w_j.

At any point in the execution of the scene the contextual information must be the same for all the governors and the scene manager. They must be constantly informed of any event that updates it. These events are: new agents joining the scene, some participating agents leaving the scene and any transition on the conversation protocol. A transition can be caused by the expiration of a timeout or by the utterance of an illocution. A fundamental aspect is that the state of the scene execution must be perceived to be the same for all the agents. That is, all of them must progress together when there is a transition in the protocol. This implies that only one agent can speak at the same time. Thus, governors and scene manager must coordinate in order to guarantee that only one of them updates the contextual information at the same time. For this purpose, they use a token passing protocol and only the agent that has the token can send a message that updates the contextual information. Furthermore, the token contains the number of events produced since the beginning of the scene. This permits governors to check if they have the complete information of all events

produced so far. Otherwise they would ask for the missing bits to the scene manager.

A participating agent can do three actions within a scene:

- it can try to utter an illocution;
- it can inform the governor that it wants to leave the scene to go to a transition; and
- it can request for scene information.

In this later case, it can ask for the list of participants within the scene, the current state and the transitions that it can reach from the scene. Furthermore, an agent is informed about changes in the list of participating agents, about messages received by the governor addressed to it, and about transitions caused by a timeout. Agents are also informed when the scene finishes.

Now we concentrate on the process that a governor follows when requested by its agent to utter an illocution in a scene conversation protocol (see figure 6.4). As the governor keeps information of all the participants in a scene along with their role, there is no need for the agent to send all the information corresponding to the sender and receiver(s) when trying to utter an illocution. If some information is left out, the governor expands the message by filling them with the information that it keeps. Table 6.3 shows how the sender and receiver fields of a received message are expanded by a governor if its agent is the agent $polki$ playing role a, and there is another agent, $polki2$, playing role b within the scene. In the case of the sender, if some information is left out, the governor expands it with the identifier and role of the agent associated to it. In the case of the receiver field, if no information is received, it is understood that the message is addressed to *all* the agents within the scene. If only the identifier of the addressee is given, the governor expands the message by adding the role that this agent is playing within the scene. This process is done before starting the verification of the message.

Once the message has been expanded, the governor checks if it is correct with respect to the scene specification and the scene execution context. An illocution sent by the agent will be correct if it matches an illocution scheme labelling an outgoing arc of the current state and it satisfies the constraints associated to that arc. Notice that within an illocution scheme we distinguish between application and binding occurrences of the variables. The governor substitutes each application occurrence in the illocution schemes labelling the outgoing arcs of the current state by the last bound values of the variable. That is, for each application occurrence the governor searches in Σ for the last bound value of the variable and it substitutes the application occurrence by the bound value. Thereafter, it applies pattern matching between the expanded illocution received from the agent and illocution schemes labelling the outgoing arcs of the current state where application occurrences have been substituted by the last bound value of the variable. If the pattern matching fails in all the cases, it means that the illocution is not correct and the agent is informed about it. If

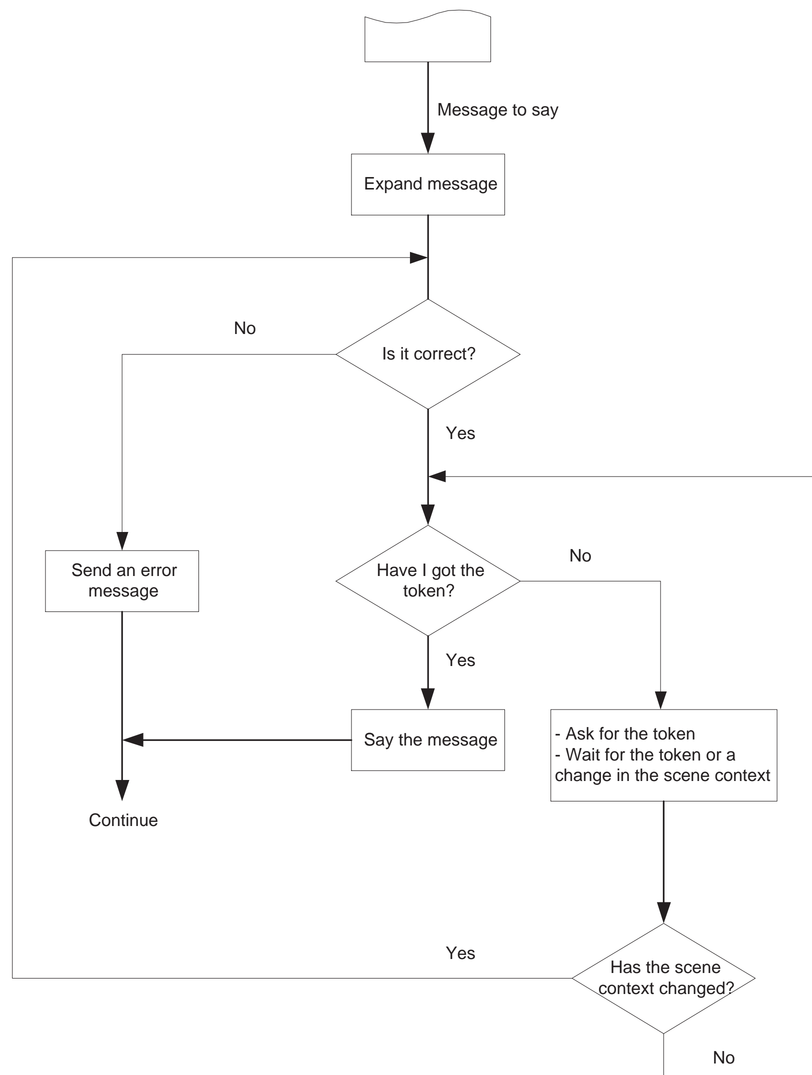


Figure 6.4: Algorithm for saying an agent message

the pattern matching succeeds, a list of substitutions for free variables in the schema is obtained and the governor checks if the arc constraints are satisfied.

In order to evaluate constraints, the variables are substituted by their actual values. On the one hand, occurrences making reference to variables in the illocution scheme of the same arc are substituted by their value in the illocution sent by the agent. On the other hand, occurrences making reference to previous binding(s) of a variable have their bound value searched for in Σ. Remember that within constraints we allow to make reference to a set of variable bindings

(see section 3.2.2 for all possibilities). Once variables have been substituted by their corresponding values, constraint expressions are transformed to postfix Polish notation [Tremblay and Sgrenson, 1985] and are evaluated using a stack. For instance, the expression (> (+ 2 3) 4) is transformed to the expression '2 3 + 4 >'. As a result, they can be easily evaluated by using a stack.

If the illocution is correct (it matches an illocutions scheme and satisfies the arc constraint) the governor tries to send it. If the governor has the token, it is allowed to send the message. The message is sent to all the governors and the scene manager because they have to update their contextual information but only the governor(s) of the receiving agent(s) forward it to their agent(s). To update the contextual information of the scene the target state of the arc becomes the current state of the scene and Σ is extended with the bindings produced in the pattern matching process. The governor confirms the agent that its illocution has been uttered by sending it a $saidSceneMessage$ message. If the governor does not have the token, it asks for the token to the agent that has got it. At this point, it can either receive the token or a message from another governor corresponding to the utterance of an illocution by another agent. In the first case, if it receives the token, it sends the message to all the governors and scene manager as explained above. Otherwise, as the context of the scene has changed it must verify whether the message is still correct and if so, it asks for the token again. Notice that the verification of the message is done before asking for the token to avoid the possibility of an agent blocking the scene when its governor has the token by continuously sending incorrect messages.

When the governor that has the token receives a message asking for it, it checks first whether it needs the token or not. If it needs the token, the request message is stored to be processed later on. Otherwise, it looks in the queue of incoming messages if there are other governors asking for the token. In this case it selects randomly one of them and it sends the token to it, except if the scene manager has asked for the token. In this later case the token is sent to the scene manager.

The following example illustrates how this works. Consider an auction scene at state w_i connected to state w_j by an arc labelled with the illocution scheme (*commit* (?*y buyer*) (!*x auctioneer*) (*bid* !*good_id* ?*price*)) and the constraint (> ?*price* !*reserve_price*). This illocution allows buyers to submit bids. Consider also that agent *John*, playing the buyer role within the scene, sends to its governor the following message to submit a bid:

$$request(c1, SaySceneMessage(commit\ (James)\ (bid\ g1\ 25)))$$

where $c1$ stands for the identifier of the conversation between the agent and its governor which corresponds to the auction scene.

When the governor receives the message the thread in charge of the communication channel parses it and adds the message to the queue of the conversation of the auction scene. Later on, the thread in charge of this conversation will read the message, it will detect that the agent wants to utter an illocution within the scene and it will expand the message. We can see that the message sent by the

Received message				Expanded Message			
Sender		Receiver		Sender		Receiver	
Agent	Role	Agent	Role	Agent	Role	Agent	Role
-	-	-	-	polki	a	all	-
-	-	all	-	polki	a	all	-
polki	-	-	-	polki	a	all	-
polki	a	-	-	polki	a	all	-
polki	-	all	-	polki	a	all	-
polki	a	all	-	polki	a	all	-
-	-	-	b	polki	a	-	b
polki	-	-	b	polki	a	-	b
polki	a	-	b	polki	a	-	b
-	-	polki2	-	polki	a	polki2	b
-	-	polki2	b	polki	a	polki2	b
polki	-	polki2	-	polki	a	polki2	b
polki	-	polki2	b	polki	a	polki2	b
polki	a	polki2	-	polki	a	polki2	b
polki	a	polki2	b	polki	a	polki2	b

Table 6.3: Expansion of messages.

agent has no information about the sender and about the role of the receiver. Then, the governor will expand the message by filling the missed information with the information it keeps. That is, with the information of its associated agent and with the role of agent $James$. If agent $James$ plays the auctioneer role within the scene, the received illocution will be expanded into the following one:

$$(commit\ (John\ buyer)\ (James\ auctioneer)\ (bid\ g1\ 25))$$

Complementary, application occurrences in the illocution scheme for submitting bids are substituted by the last bound value of variables. Looking at illocution scheme presented above, there are two application occurrences: $!x$ and $!good_id$. Therefore, it is looked in Σ for their last bound value. If the last bound value of variables x and $good_id$ are $James$ and $g1$ respectively, the illocution scheme presented above is transformed into the following one:

$$(commit\ (?y\ buyer)\ (James\ auctioneer)\ (bid\ g1\ ?price))$$

Then, the governor applies pattern matching between the expanded illocution from the agent and the illocution scheme where application occurrences have been substituted by the last bound value of the variables. In this case the pattern matching succeeds and the following list of substitutions is obtained: $[?y/John, ?price/25]$. Notice, that if the last bound value of variable x would be different of $James$ or the last bound value of variable $good_id$ would be different than $g1$, the pattern matching would have failed. Finally, before trying to send

the illocution, the constraints associated to the arc must be evaluated. In order to evaluate the constraint, its variables must be substituted by their es must be substituted by their e occurrence ?*price* makes reference to the value of the variable in the illocution sent by the agent and the occurrence !*reserve_price* to the last bound value of the variable, which is search for in Σ. The constraint is satisfied if the last bound value of variable *reserve_price* is lower than 25. If this is the case, the message is correct and the governor would try to send the message. With this aim it would ask for the token, and if it receives the token, it will send the illocution to all the governors and the scene manager making the scene evolve. Only the governor of agent *James* would pass the message to its agent. As a consequence of the illocution utterance the current state of the scene will evolve to w_j and Σ is extended with $\sigma_{w_i wj}$, the bindings produced by the utterance of the illocution, namely $[?y/John, ?price/25]$.

Another important issue within a scene execution is when there is a timeout. As soon as the scene evolution reaches a state where there is an outgoing arc labelled with a timeout, governors evaluate the timeout expression and start the timeout countdown. Timeout expressions are evaluated as constraint expressions. Thus, variables appearing in the expressions are substituted by their bound values to evaluate the expression using a stack. Notice, that in the case of timeouts all variable occurrences must correspond to application occurrences. This process is done independently by each governor of the scene and the countdown is done in a newly created thread. If the governor receives a message of its agent before the timeout expires, it proceeds as explained above. Otherwise, if the timeout expires, this means that its agent can not utter any illocution and must coordinate with the scene manager and the other governors to know if all of their timeout countdowns have expired or if there is some agent that uttered an illocution on time. The coordination in this case is lead by the scene manager. When a governor timeout expires it sends a message to the scene manager. If the scene manager receives a message from each governor confirming timeout expiration, it asks for the token and sends a message to all the governors informing that the scene state evolves as a consequence of the timeout expiration.

Scene managers are also in charge of authorising agents to join or leave scene executions. On the one hand, requests for joining the scene are received from the institution manager which is in charge of transitions. On the other hand, when an agent intends to leave a scene, it must send the governor a message *movetoTransition* informing the governor to which transition it wants to go to. If the agent can move to the requested transition (if there is an arc in the performative structure from the scene to the requested transition labelled by the agent's role) the governor informs the scene manager that its associated agent wants to leave the scene. In both cases the scene manager informs all the governors about the requests and the roles of the agents waiting to join or leave the scene. Thereafter, when the scene reaches a state where the action can be performed the governor owning the token sends it to the scene manager. This occurs, when the scene execution reaches an access or exit state for the role of an agent waiting for joining or leaving the scene. Upon the reception of the

token, the scene manager blocks the scene execution and authorises agents to join or leave except if the restrictions on the maximum and minimum agents per role would be violated. When an agent is authorised to join the scene, the scene manager sends all the contextual information about the current execution to its governor, namely the current state, Σ and the list of participants. The governor keeps the received information and informs its agent that it has joined the scene, and about the current state and the current participants within the scene. When an agent is authorised to leave the scene, it goes to the selected transition. Then, the governor informs the agent that it has moved from the scene to the selected transition. In both cases the scene manager informs all the governors about the agents that have joined or left the scene and governors update the contextual information and inform their associated agents.

6.3.3 Transition management

Transitions are a kind of routers within the institution performative structure whose dialogues refer to the scenes that agents within them can reach. The institution manager is in charge of managing the transitions, because it knows all the current executions of the different scenes in the performative structure, and which is the scene manager of each one of them. Furthermore, it is in charge of authorising agents to move to current scene executions and of creating new ones.

The outgoing arcs that an agent reaching a transition can follow depend on the arcs' labels. Since incoming and outgoing arcs of a transition are labelled with pairs of agent variable and role identifier, an agent can only leave a transition through those arcs labelled with the same agent variable than the one in the arc through which it reached the transition. Notice that we allow agents to change its role when traversing a transition before joining target scenes. Therefore, if an agent reaches the transition through an arc labelled with $(x\, r)$ its possible paths are those outgoing arcs whose label contain a pair $(x\, r')$. Furthermore, the type of each outgoing arc defines if an agent following the arc will join to one, some or all the current executions of the target scene or if the agent will join a newly created execution of the target scene.

Complementarily, the type of a transition determines whether the agent will follow *all* or must select *one* of its possible paths and if the agent has to synchronise with other agents. Remember that there are two types of transitions: *Or*, and *And*. Agents reaching an *Or* transition must choose only one of its possible paths to follow, while agents reaching an *And* transition will follow all its possible paths. Moreover, *And* transitions force agents to synchronise before leaving the transition.

When an agent arrives to a transition, it can ask which scenes it can go by sending an *accesScenes* message to its governor. The governor passes the message to the institution manager, that taking into account the path that the agent followed to the transition, the performative structure specification and the current scene executions, provides the paths that the agent can follow. Concretely, each path contains:

- the type of the target scene of the arc;
- the type of the arc;
- the role that the agent will play in the target scene; and
- the current scene executions of the target scene.

For those arcs of type *new*, an empty list is sent in the last field because if the agent follows that arc it will go to a newly created scene execution of the target scene. When the governor receives the information from the institution manager, passes it to the agent.

From its possible destinations the agent must request which paths to follow by sending a *movetoScenes* message to its governor. The message must contain per selected path the type of the target scene, the type of the arc, the role that the agent is going to play in the destination scene(s), and a list of scene executions. The last field is only necessary when the arc is of type *one* or *some*; containing only one scene execution identifier if the arc is of type *one* and a list of executions identifiers if the arc is of type *some*. In the case of arcs of type *new* or *all*, this last field must be an empty list as in the first case, the agent will go to a newly created scene execution and in the second case, the agent will go to all the current executions of the target scene.

The governor passes the message to the institution manager which analyses if the agent request is correct. For instance, that it does not select more than one path to follow in an *Or* transition, that it does not select more than one execution to go to in an arc of type *one* or that the identifiers of the scene executions that it requests to go are correct. The institution manager analyses the agent request taking into account the type of the transition, the arc that the agent followed to reach the transition, the transition outgoing arcs and the current executions of each target scene. If it is not correct, the agent has its request refused.

The last point to take into account before allowing an agent to move is whether it has to synchronise with other agents or not. Agents are forced to synchronise when they reach a transition of *And* type. In the case of an *Or* transition agents reach and leave the transition by themselves following one of its possible paths, without synchronising with other agents. The synchronisation is undertaken by the institution manager as it knows all the agents within the transition and the scene executions that each agent has requested to go to. Notice that agents appearing in a conjunction in an outgoing arc of the transition must go to the same scene executions and then, which agents are synchronised depends in some cases on which scene executions they request to go to. This corresponds to the cases where the conjunction appears in an arc of type *one* or *some*. If there are no conjunctions in an arc of type *one* or *some*, synchronisation is accomplished taking into account the order in which agents reach the transition. In this manner, the first agents reaching the transition for each incoming arc are synchronised. If there is a conjunction in an arc of type *one* or *some* agents are synchronised with the agents that want to go to the same scene execution(s). For

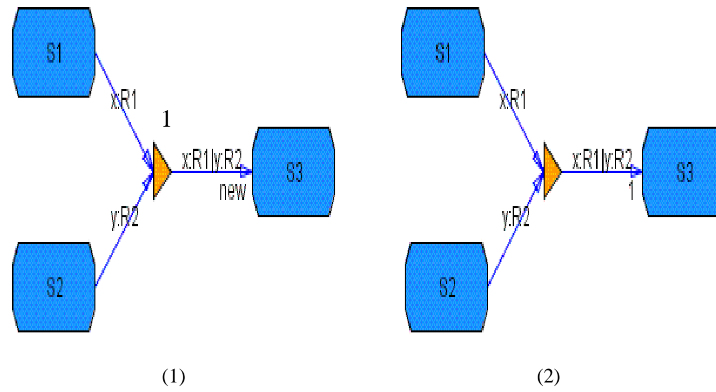


Figure 6.5: Examples of transitions.

this purpose, the institution manager keeps which scenes each agent within the transition wants to go to and it uses this information to synchronise the agents. When a new agent reaches a transition requesting which scene executions it wants to go to, the institution manager looks if the agent can be synchronised with some of the agents on wait. If so, the institution manager synchronises the agents and authorises them to start moving to the requested scene executions. We want to remark that when the institution manager has different possibilities for synchronising the new agent, it always chooses the agents that first reached the transition. (the agents that have been waiting for longer). While an agent is waiting for being synchronised it can change its target scenes by sending a new message $movetoScenes$ to its governor.

Once an agent has correctly selected where to move to and after being synchronised, if necessary, it can start moving. Movements are done asynchronously, and agents are incorporated into each requested scene execution, as soon as possible, without taking into account other destinations. From the set of movements that an agent has requested, we differentiate between the movements to active scenes and the movements to new scenes. When the movement is to an active scene, the institution manager communicates the scene manager of the target scene execution that there is an agent or a group of agents waiting to join the scene as well as their role(s). The scene manager will allow the agent(s) to join the scene, as soon as the scene execution reaches an access state for that role(s). When the movement is to a new scene execution, this is created by launching a scene manager for it, and the agent(s) is incorporated into it..

Figure 6.5 depicts two examples of scene connections mediated by an *And*

transition. In (1), agents coming from scenes $s1$ and $s2$ are synchronised in order to start a new execution of scene $s3$, while in (2) they are synchronised to go to *one* of the current executions of scene $s3$. Consider that an agent $a1$ playing the $R1$ role reaches the transition from $s1$; that there are no agents waiting in the transition; and that there are three executions of $s3$ denoted by $i1\#s3$, $i2\#s3$ and $i3\#s3$. Once in the transition the agent can request which scenes to go by sending the following message to its governor:

$$request(ci, accessScenes)$$

where ci stands for the conversation between the agent and the governor associated to the transition.

The governor, after communicating with the institution manager, answers the agent with the following message (the first one corresponds to (1) and the second one to (2)):

$$(1)\ inform(ci, currentAccessScenes((s3, new, R1, nil)))$$

$$(2)\ inform(ci, currentAccessScenes((s3, one, R1, (i1\#s3, i2\#s3, i3\#s3))))$$

In the first case, as the destination will be a newly created scene execution of the target scene, no information is sent about the current executions of $s3$. In the second case, as the arc is of type *one* the agent is informed about all the current executions of $s3$. After receiving the information, the agent must request where it wants to go to by sending a $movetoScenes$ message to its governor. In the first case, it has only one option while in the second one, it must choose one of the current executions to join. Consider that in each case it sends the following messages to the governor:

$$(1)\ request(ci, movetoScenes((s3, new, R1, nil)))$$

$$(2)\ request(ci, movetoScenes((s3, new, R1, (i1\#s3))))$$

Afterwards, the governor passes the message to the institution manager which analyses the agent request. Since the agent request is correct in each case, the institution manager keeps it and the agent waits to be synchronised. In the first case, the agent will be synchronised with the first agent coming from $s2$ to enter a newly created execution of $s3$. At this aim the institution manager will launch a scene manager for the new scene execution to which agents will be incorporated.

In the second case, the agent must wait until the arrival of an agent from $s2$ intending to go to the very same scene execution, that is to execution $i1\#s3$. Notice that if an agent intending to go to another scene execution arrives from $s1$, this agent will not be synchronised with agent $a1$. However if an agent arriving from $s2$ requests for joining $i1\#s3$, it will be synchronised with $a1$ and both will be authorised to move to the selected scene execution. After being synchronised the institution manager will inform the scene manager of execution $i1\#s3$ which

will allow agents to join the scene, as soon as, it will reach an access state for their roles, that is, for roles $R1$ and $R2$.

We have to take into account that some of the requested movements of an agent can fail since an agent might not join a requested scene execution before it finishes. This can occur because the scene did not reach an access state for the agent role before finishing or because there were already in the scene the maximum number of agents per its role and therefore it was not allowed to join the scene. If the arc to the scene is of type *all* or *some*, the agent is informed and it is moved to the other selected executions. But, if the arc is of type *one* this path is considered failed and the agent is asked to select another scene execution to go.

6.3.4 Norm management

Norms model the consequences of agent's actions within scenes. These consequences are expressed in terms of obligations that agents must fulfil later on. Norms contain the actions that will provoke their activation, the obligations that agent will have and the actions that agents must carry out in order to fulfil the obligations. As we are in dialogic institutions the actions are expressed as pairs of a scene and an illocution scheme. Norms have the following schema:

$$(s_1, \gamma_1) \wedge \ldots \wedge (s_m, \gamma_m) \wedge e_1 \wedge \ldots \wedge e_k \wedge$$
$$\wedge \neg(s_{m+1}, \gamma_{m+1}) \wedge \ldots \wedge \neg(s_{m+n}, \gamma_{m+n}) \rightarrow obl_1 \wedge \ldots \wedge obl_p$$

where $(s_1, \gamma_1), \ldots, (s_{m+n}, \gamma_{m+n})$ are pairs of scenes and illocution schemes, $e_1, \ldots e_k$ are boolean expressions over illocution schemes variables, $\neg$ is a defeasible negation, and $obl_1, \ldots, obl_p$ are obligations. The meaning of these rules is that if the illocutions $(s_1, \gamma_1), \ldots, (s_m, \gamma_m)$ have been uttered, the expressions $e_1, \ldots, e_k$ are satisfied and the illocutions $(s_{m+1}, \gamma_{m+1}), \ldots, (s_{m+n}, \gamma_{m+n})$ have *not* been uttered, the obligations $obl_1, \ldots, obl_p$ hold. Therefore, the rules have two components, the first one is the causing of the obligations to be activated (for instance winning a Dutch auction round by saying 'mine', generates the obligation to pay) and the second is the part that removes the obligations (for instance, paying the amount of money due for the round which was won).

Remember that norms are specified in ISLANDER by three components, the antecedent, the defeasible antecedent and the consequent. The antecedent contains the list of actions that provoke the activation of the norm, the consequent contains the list of obligations that the agent will have when the norm is activated and the defeasible antecedent contains the actions that an agent must do in order to fulfil the obligations.

Governors keep, at every moment, the pending obligations of their associated agent and they check whether agent interactions modify them. This can happen as a consequence of the activation of some norms or as a consequence of the fulfilment of some of the agent pending obligations. On the one hand, the governor must check whether the illocutions uttered and received by its associated agent satisfy any of the norm antecedents implying that the agent has acquired new

obligations. On the other hand, the governor must check whether the illocutions uttered and received by its agent satisfy the defeasible antecedent of an activated norm in order to remove the obligations.

Our approach is to manage norms as a rule based system. In order to construct the rule base, each institutional norm is divided into two rules, one for the activation of the norm and another one for the fulfilment of obligations. The facts of the system are the illocutions uttered and received by the agent. Notice that not all the rules are active at every moment. The rules that check norms' activation are always in the rule base but the rules corresponding to the fulfilment of obligations are added and deleted dynamically from the rule base when a norm is activated or when obligations are fulfilled respectively. A norm N_i which follows the schema presented above is divided into the two following rules:

$$R1_i : (s_1, \gamma_1) \wedge \ldots \wedge (s_m, \gamma_m) \wedge e_1 \wedge \ldots \wedge e_k \rightarrow$$
$$assert(obl_1 \ldots obl_p) \wedge addRule(R2'_i, RB)$$

$$R2_i : (s_{m+1}, \gamma_{m+1}) \wedge \ldots \wedge (s_{m+n}, \gamma_{m+n}) \rightarrow$$
$$retract(obl_1 \ldots obl_p) \wedge dropRule(R2_i, RB)$$

The first rule corresponds to the norm activation. The meaning of $R1_i$ is that if illocutions $\gamma_1 \ldots \gamma_n$ have been uttered in the corresponding scenes and $e_1 \ldots e_k$ are satisfied, then obligations $obl_1 \ldots obl_p$ are added to the set of agent pending obligations and a rule to check the obligations fulfilment is added to the rule base. Notice that illocution schemes on norm definitions contain variables whose scope is the complete norm. Hence that the bindings of these variables must be taken into account in the rule of the second type added to the rule base. Thus, $R2'_i$ is a particularization of $R2_i$ where variables are replaced by their bound value.

The second rule checks whether norm obligations are fulfilled. The meaning of $R2_i$ is that if illocutions $\gamma_{m+1} \ldots \gamma_{m+n}$ have been uttered in the corresponding scenes, then obligations $obl_1 \ldots obl_p$ are eliminated from the set of agent pending obligations and the rule is removed from the rule base. Notice that at a certain moment there can be more than one rule of the second type from the same norm in the rule base, each one corresponding to a different activation of the norm. Governors only need to add to their rule bases the first type of rules because the second type will be added and removed dynamically in the rule base as new obligations are acquired or fulfilled.

In order to manage the rules we have opted for using the Java Expert System Shell (JESS) [JESS, URL]. JESS is a rule engine and scripting environment which permits the creation and management of rule-based systems from JAVA programs. In order to check which rules can be fired, it has implemented a *rete* algorithm which is a well known forward checking algorithm [Forgy, 1982].

The governor has a thread devoted to manage its connection to JESS. On the one hand, this thread will be in charge of adding the rules and facts into JESS

and to run the *rete* algorithm. On the other hand, this thread will be informed by JESS whenever a rule is fired. When the institution specification is loaded each norm in ISLANDER is transformed into a rule in JESS and added into JESS. That is, rules where the antecedent is the antecedent of the norm, and as a consequent they have the obligations expressed in the norm consequent, and a rule definition for checking the fulfilment of the obligations. Thus, when one of these rules is fired a new rule will be added to the rule base for checking the fulfilment of obligations.

Taking into account that few of the illocutions that agents utter can activate a norm or fulfil agent obligations to add all the illocutions that an agent sends or receives to JESS would make the *rete* algorithm very inefficient. Only the uttered illocutions which match an illocutions scheme in a norm should be added to JESS. For this purpose, for each scene it is marked which illocution schemes can be matched by illocutions which can also match some of the illocution schemes appearing in the norms. When, an illocution matching one of the marked illocutions where the agent associated to the governor is uttered within a scene, the thread handling that conversation passes it to the thread in charge of JESS.

The thread in charge of JESS works as follows:

[**Step 1**] Waits for a new illocutions from one of the threads in charge of the conversations.

[**Step 2**] Adds the new fact into JESS.

[**Step 3**] Runs the rete algorithm.

[**Step 4**] Goes back to step one.

At each execution of the rete algorithm JESS checks if the new facts fire any rule and it stops when no more rules can be fired. Whenever a rule is fired, JESS informs the thread which informs the agent and modify the list of agent pending obligations. On the one hand, when a rule of the first type is fired the agent is informed about the new obligations that it has acquired. On the contrary, when a rule of the second type is fired the agent is informed about the fulfilled obligations.

6.4 Development of agents for electronic institutions

Until now we have focused on how to develop infrastructures for electronic institutions but another fundamental issue is the development of agents which can participate in the institution. Remember that in institutions' formalisation we differentiate between the internal and external roles. The internal roles are played for what we call the staff agents that represent the electronic counterpart of the institution workers in human institutions. Since the institution delivers

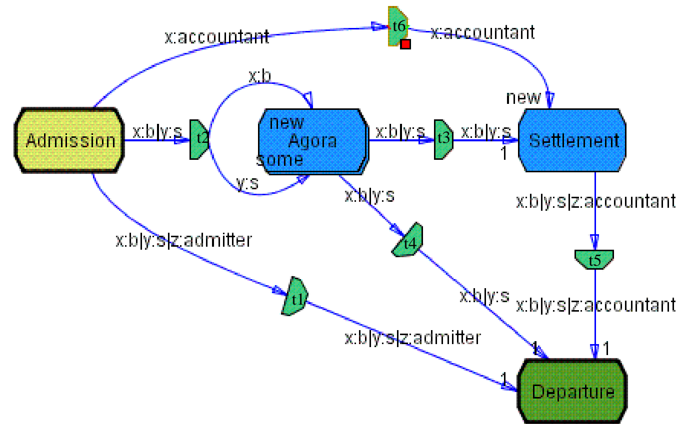


Figure 6.6: Performative structure of the agoric market.

its services and duties to staff agents a complete development of them is needed in order to run the institution. However, the process presented here to develop agents for an institution can be used to develop indistinctly a staff or an external agent.

The specification defines what the agents within an institution are allowed to do but no information is given on how agents have to take their decisions. Hence, agents can not be automatically generated from the specification. But agent skeletons can be obtained from the institution specification. And yet, the work of agent designers will be to fill up the parts that can not be extracted from the institution specification. For this purpose, they must define *when* to speak and *what* to say, which conversations to join to and which information from the received illocutions must be kept in the agent's knowledge base to be used in further decisions.

The purpose of this section, which is a summary of [Vasconcelos et al., 2002b, Vasconcelos et al., 2003][2], is to show how agent skeletons for an electronic institution can be first synthesised from the institution specification and how they can be later on customised by engineers. Next, in section 6.4.1 we present the agoric market which we use as an example throughout the section. In section 6.4.2, we explain a logical formalism for electronic institutions in Prolog, section 6.4.3 describes how agent skeletons can be synthesised from the logical representation of an institution and section

[2]The main ideas of this section correspond to Wamberto Vasconcelos and he has also developed the software components explained here. I want also to thank him and the rest of the authors of both paers for authorising us to use material from them.

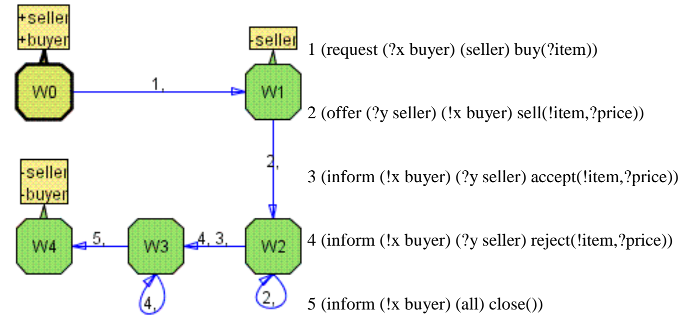


Figure 6.7: Simple Agora Room Scene

6.4.4 details how the skeletons can be customised by agent developers. Finally, in section 6.4.4 we present a generic seller agent for the agoric market.

6.4.1 Agoric Market

As an example we use a virtual agoric market that agents enter to buy and sell goods. Figure 6.6 depicts the performative structure of the market. The main scene is the agora, depicted in figure 6.7, where agents interact for buying and selling goods. Before agents can take part in the agora they have to be admitted and after the agora room scene is finished, buyers and sellers must proceed to settle their debts. Within an agora room (see figure 6.7) an agent willing to acquire goods interacts with a number of agents intending to sell such goods. This agora scene has been simplified – no auctions or negotiations are contemplated. The buyer announces the goods it wants to purchase, collects the offers from sellers (if any) and chooses the best (cheapest) of them.

6.4.2 Logical formalism for Electronic institutions

Once the institution has been specified it is translated from ISLANDER into a logical formalism [Vasconcelos et al., 2002a] implemented in Prolog [Apt, 1997] making the representation computer-processable. This makes easier to synthesise our simple agents, as we shall see below.

We show in Figure 6.8 our Prolog representation for the agora room scene depicted in Figure 6.7. Each component of the formal definition has its corre-

```
roles(agora,[buyer,seller]).           states(agora,[w0,w1,w2,w3,w4]).
initial_state(agora,w0).               final_states(agora,[w4]).
access_states(agora,buyer,[w0]).       access_states(agora,seller,[w0]).
exit_states(agora,buyer]).
exit_states(agora,buyerler,[w1,w4]).
theta(agora,[w0,request(B:buyer,all:seller,buy(I)),w1]).
theta(agora,[w1,offer(S:seller,B:buyer,sell(I,P)),w2]).
theta(agora,[w2,offer(S:seller,B:buyer,sell(I,P)),w2]).
theta(agora,[w2,inform(B:buyer,S:seller,accept(I,P)),w3]).
theta(agora,[w2,inform(B:buyer,S:seller,reject(I,P)),w3]).
theta(agora,[w3,inform(B:buyer,S:seller,reject(I,P)),w3]).
theta(agora,[w3,inform(B:buyer,S:seller,close()),w4]).
```

Figure 6.8: Representation of Agora Room Scene

sponding representation. Since many scenes may coexist within one electronic institution, the components are parameterised by a scene name (first parameter). The arcs among scene states and their labels are represented together in `theta`/2 where the second argument holds a list containing the directed edge as the first and third elements of the list and the label as the second element.

```
scenes([admission,agora,settlement,departure]).
transitions([t1,t2,t3,t4,t5,t6]).
root_scene(admission).             output_scene(departure).
arc([admission,w3],p_1,t1).        arc(t1,p_1.1,[departure,w0]).
arc([admission,w3],p_2,t2).        arc(t2,p_2.1,[agora,w0]).
arc(t2,p_2.2,[agora,w0]).          arc([agora,w1],p_3,t3).
arc([agora,w4],p_3,t3).            arc(t3,p_3.1,[settlement,w0]).
arc([agora,w1],p_4,t4).            arc([agora,w4],p_4,t4).
arc(t4,p_4.1,[departure,w0]).      arc([admission,w3],p_6,t6).
arc(t6,p_6.1,[settlement,w0]).     arc([settlement,w3],p_5,t5).
arc(t5,p_5.1,[departure,w0]).
```

Figure 6.9: Representation of Agoric Market electronic institution

Any scene can be conveniently and economically described in this fashion. In Figure 6.9 we present a Prolog representation for the agoric market institution. Of particular importance are the arcs connecting scenes: these are represented as `arc`/3 facts storing the first argument of which holds (as a sublist) an exit state of a scene, the second argument holds the predicate (constraint) p_i which enables the arc, and the third argument is the destination transition. Each p_i contains which roles can progress through the arc. Another `arc`/3 shows arcs leaving a transition and entering an access state of a scene. Notice that the representation of the performative structure is a little bit different from the one that we have presented in chapter 3. Concretely, connections are done from scene exit states to transitions and from transitions to scene access states. Furthermore, norms are not represented in the current version of the logical formalism.

6.4.3 Synthesis of Agents

The basic idea for synthesising agents is to automatically extract from an electronic institution an account of the behaviours agents ought to have. This sim-

plified account is called a *skeleton*: it provides the essence of the agents to be developed. A simple way to synthesise agents from our institutions is introduced in [Vasconcelos et al., 2002a]. We devised a means to use the logical representation of the electronic institution in order to obtain a set of Horn clauses which capture the behaviours for the agents participating in the institution. The synthesis obtains, for the roles of each scene, a set of Horn clauses which represent the connections among the states and the events, *i.e.*, sending or receiving messages, associated with these edges. Engineers willing to develop agents to perform in electronic institutions could then be offered a skeleton which would be gradually augmented into a complete program. Depending on the way skeletons are represented, semi-automatic support can be offered when augmenting them into more complex programs.

A skeleton defines all the basic behaviours agents should possess to successfully perform in the institution they are designed for. Our skeletons are simple logic programs with very limited functionality: they store the current state of the computation, and are able to move on to a next state, given certain conditions. However, there might be states of the computation from which more than one next state is possible. That is, situations where an agent can do different actions. For instance, there are states of a scene where an agent can utter different illocutions, or when a scene reaches an exit state for its role an agent might choose between leaving the scene or continuing participating within it. When a rational agent follows an institution, which is the next action to do should be resolved by formal reasoning and decision-making procedures. The augmenting process which skeletons undergo is aimed at "filling in" such capabilities. Reasoning and/or decision-making procedures have to be appropriately added to the initial skeleton, yielding more sophisticated agents that conform to the institution from which they were extracted. Furthermore, any variation to be performed by the components (such as the customisation of messages) is not specified in the institution specification. If, for instance, a message offering an item is to be sent, the actual item which is offered is to be defined by whichever agent actually participates in the institution. This variability is another capability that ought to be added to the initial skeleton.

We show in Figure 6.10 some of the clauses synthesised from the electronic institution of Figure 6.6, represented as in Figures. 6.8 and 6.9. The top clauses depict the agora scene. The bottom clauses are the transitions among scenes. Additional predicate definitions are required for message exchange and these are inserted at a later stage. An agent whose predicates are all defined is a completely operational and executable Prolog program which captures the behaviours within an electronic institution.

The clauses define predicate `s/1` which uses a list to represent the current state. The list consists of the name of the scene, the name of the state in the graph and the role of the agent. Depending on the role of the agent, a suitable action `send/1` or `rec/1`, to send and receive a message, respectively, is chosen for the clause. By using the clauses with the standard SLDNF resolution mechanism [Apt, 1997] we get all possible behaviours of the agents in the electronic

```
s([agora,w0,buyer]):-
  send(request(B:buyer,all:seller,buy(Item))),
  s([agora,w1,buyer]).
s([agora,w0,seller]):-
  rec(request(B:buyer,all:seller,buy(Item))),
  s([agora,w1,seller]).
...
s([agora,w3,seller]):-
  rec(inform(B:buyer,S:seller,reject(Item,Price))),
  s([agora,w3,seller]).
s([admission,w3,seller]):- holds(p1),s([t1,seller]).
...
s([t5,buyer]):- holds(p5.1),s([departure,w0,buyer]).
```

Figure 6.10: Synthesised Agent from Agoric Market institution

institution.

6.4.4 Customising Synthesised Agents

The clauses synthesised from the logic representation of an institution describe all possible behaviours an agent may have. Because it is an exhaustive process, all scenes, edges, transitions and roles are considered. However, if we were to use the same clauses to define agents which would enact an institution, they would all have precisely the same behaviours. Although this might be desirable at times, we also want to offer means for designers to add *variability* to the agents synthesised.

This initial skeleton is then customised in different ways by the user. We are able to represent a comprehensive repertoire of program manipulation operations organised in the following three categories (in increasing order of complexity of captured programming expertise):

- *Program editing* – operations such as insert/delete an argument in a predicate, insert/delete goal in a clause, insert/delete clause in a program, and so on.
- *Electronic institution editing* – operations to *restrict* the clauses to specific scenes, states of a scene, transitions and roles. Such operations take into account the inter-dependence of concepts within the institution; for instance, if an agent has access to scene $\mathbf{S}_1$ then it may also need to have access to scene $\mathbf{S}_2$; if the user tried to restrict the clauses to scene $\mathbf{S}_1$, a message would be issued.
- *Program techniques* – insertion of extra functionalities with a coherent meaning/purpose, such as pairs of accumulators to carry values around, building recursive data structures, and so on [Sterling and Shapiro, 1994].

These operations require user intervention in order to be properly applied. Users must determine where an argument is to be inserted, which transition, scene, or role is to be removed from the program being built, and so on. Our environments

also offer the means to perform manual editing: the users are presented with the code for the program in a text editor and they can alter the program in whichever way wanted.

We show in Figure 6.11 the first two clauses of the synthesised agent with an example of the kinds of customisation via augmenting that users are allowed to perform within the environment. Starting with the synthesised clauses of Figure 6.10 the user gradually adds features to the agent's capabilities. We show the added parts underlined. The first modification inserts a programming technique which carries a `Stock` data structure around as program execution proceeds; this data structure is employed to obtain, via predicate `chooseItem`/2, the value of `Item` in the first clause. The definition for `chooseItem`/2 must be supplied. The second modification concerns the addition of another technique to assemble a data structure `Msgs`. This data structure stores the messages sent and received, and is updated by means of calls to predicate `update`/3 (which should also be supplied). The environment ensures that arguments are consistently inserted, and the user must provide suitable definitions for any auxiliary predicates. The original set of behaviours of the synthesised agent is preserved in our extended program above. Ideally this should always happen, ensuring that agents will perform correctly and efficiently/intelligently.

```
s([agora,w0,buyer],Stock,Msgs):-
  chooseItem(Stock,Item),
  send(request(B:buyer,all:seller,buy(Item))),
  updateMsgs(send,Msgs,buy(Item),NewMsgs),
  s([agora,w1,buyer],Stock,NewMsgs).
s([agora,w0,seller],Stock,Msgs):-
  rec(request(B:buyer,all:seller,buy(I))),
  updateMsgs(rec,Msgs,buy(Item),NewMsgs),
  s([agora,w1,seller],Stock,NewMsgs).
...
```

Figure 6.11: Augmented Agent

A Generic Seller Agent

When we customise our seller agents to deal with their pricing policy, we define the functions which implement the respective policies and leave a slot with the possible choices *greedy* or *considerate*. Depending on the choice taken, the distinct policies are incorporated. We can also pursue the continuum alternative and have a slot for the profit margin which will be a numeric value between 0 and 100 to be used by the seller agents when assigning prices to items. We can be very specific and independently carry out the alterations which will define the greedy and considerate policies, but we have noticed that these are very similar, the only distinction being the percentage of profit to be added to the price. We show in Figure 6.12 the clause of the `GenericSeller` agent, where the pricing is established as well as the definition of one of the auxiliary predicates and design options. The `s`/3 definition shows the edge $w_1 \rightarrow w_2$ when the seller agent responds to a buyer request: the actual request `request:buy(Item)` is retrieved

from the messages received `Msgs`, the price of `Item` is established via predicate `pricing`/2, the offer is sent to the buyer agent, the messages sent/received are updated via `updateMsgs`/4 and finally the seller agent moves to state w_2. Predicate `retailPrice`/2 maps each `Item` (first argument) to its suggested retail price `RPrice` (second argument).

```
s([agora,w1,seller],Stock,Msgs):-
  member(request:buy(Item),Msgs),
  pricing(Item,Price),
  send(offer(S:seller,B:buyer,sell(Item,Price))),
  updateMsgs(send,Msgs,offer:sell(Item,Price),NewMsgs),
  s([agora,w2,seller],Stock,NewMsgs).

pricing(Item,Price):-
  retailPrice(Item,RPrice),
  $greed(Profit),
  Price is RPrice + (RPrice * Profit).

designOption(predicate:greed/1,[greedy:greed(40),considerate:greed(10)]).
```

Figure 6.12: Fragment of `GenericSeller` Agent

Predicate `pricing`/2 calculates the `Price` of `Item` but it requires the definition of predicate `greed`/1 (marked with a "`$`" explained below) which obtains the profit margin the agent is to adopt. The distinction between a greedy and a considerate seller agent lies in the definition of `greed`/1. Both the continuum and the discrete possibilities can be exploited with suitable definitions of `greed`/1.

The `designOption`/2 predicate highlights that `greed`/1 is yet to be defined. When the user marks a programming construct with "`$`" our programming tool prompts her to specify what the construct is expected to be and what values it may have. This is then represented in the program itself via predicate `designOption`/2: its first argument states that a predicate `greed`/1 awaits definition and its possible definitions are represented as a list (second argument of `designOption`/2) of pairs *`Label`*:*`Definition`*. A more informative label, such as `greedy` and `considerate`, can thus be associated to a definition. The labels are used to automatically synthesise an interface to the parameter-tuning of our prototypes.

When the different type of agents has been customised, a prototype of MAS can be defined. A prototype of a MAS consists of an institution and agents to enact it. These agents have been synthesised from the institution (or from parts of it) and customised later on by the designer. Prototypes are defined as collections of *populations of agents*. Designers select from the programs obtained during the customisation stage those that will enact the institution and how many of each should make up the prototype.

In order to simulate institutions a distributed simulation platform for electronic institution has been developed. This environment permits to select populations of customised agents and simulate a prototype of the electronic institution. This proof-of-concept platform, developed in SICStus Prolog [SICS, 2000], simulates an electronic institution using a number of administrative agents, im-

plemented as independent processes, to oversee the simulation. These administrative agents look after the customised agents taking part in the institution which interact via a blackboard architecture, using the SICStus Linda tuple space [Carriero and Gelernter, 1989, SICS, 2000]. Once the simulation has been run the user is informed about its results. After analysing the results, the user can modify some of the decisions taken and then, run new simulations. This process gives rise to a virtuous lifecycle, as reported in [Vasconcelos et al., 2002a].

6.5 Conclusions

In this chapter we have presented an infrastructure for electronic institutions. We defend that the execution of open multi-agent systems require an infrastructure that must check that participating agents do not violate the rules. For this reason we have developed a social layer middleware on top of the JADE platform. The social layer middleware is in charge of allowing agents to participate in the institution but checking that they do not violate institutional rules. The agents of the social layer coordinate to guarantee the correct execution of an institution.

The most important agent of the social layer is the *governor* because each participating agent is connected to a governor to have their interaction mediated with the rest of the agents. The agent requests to the governor any information it wants about the institution or for carrying out actions. In the case of information, the governor sends it to the agent if it is authorised to receive it. In the case of actions, the governor analyses first if the action is correct with respect to the institution specification and the current execution. If so, it tries to do it in behalf of the agent and it informs the agent if it succeeds or not. Furthermore, participating agents are informed by their governors about those events in the institution that they need to know in order to participate in it.

The social layer middleware is generic in the sense that it can be used in different institutions. In order to achieve that, the agents composing the social layer are able to load XML specification of institutions as generated by the ISLANDER editor. Thus, there is no need to develop a new infrastructure for each specified institution saving effort and time to institution designers. We believe that this is an important step forward on the development of multi-agent systems for which much of the effort has been spent on the development of infrastructures.

In the last part of the chapter, we have shown how agents for electronic institutions can be developed. Firstly, agent skeletons can be obtained from the logical representation of an institution. Then, the agent designer can develop an agent customising the skeleton. That is, the designer extends the skeleton with the decision making mechanisms.

Chapter 7

Applications

The purpose of this chapter is to illustrate practically how to specify electronic institutions. Specifically we introduce two institutions: the auction house federation and the conference centre[1]. Firstly, in section 7.1 we focus on the auction house federation, while in section 7.2 we focus on the conference centre.

7.1 Auction house federation

As a first example we introduce the auction house federation institution designed and developed within the MASFIT project [MASFIT, URL]. The goal of the MASFIT project is to develop a system that allows software agents to participate in real fish markets in the same conditions as human buyers. Nowadays, some of the real fish markets have a centralised software system which controls all the processes occurring within the auction house. The system permits the registration of lots delivered to the fish market by the fishermen, has control on which buyers are taking part in the auction, auctions the fish after receiving an order from the auctioneer, and keeps track of the result of each round. In order to allow buyer agents to participate in the fish markets we have defined an electronic institution which can be connected to the software running at the auction houses.

From the point of view of the IIIA this project is a continuation of the fish market project [FishMarket, URL]. The fish market project was devoted to the design and development of an electronic version of the real fish markets, as it is thoroughly described in [Noriega, 1997, Rodríguez-Aguilar, 2001]. The differences among the implemented version and the real fish market, the necessity of connecting the institution to the software systems in the real auction houses and the possibility of having several real fish markets connected to the institution has motivated us to design and develop a new electronic institution, although the previous experience on the design and development of an electronic version

[1] Appendix A presents the complete specification of both institutions in the ISLANDER language.

of the fish market has been very valuable during the project. Thus, we have designed and developed an electronic institution which permits software buyer agents to participate in the real auctions in the same conditions as human buyers. That is, buyer agents receive the same information as human buyers in the real auction house and they have the same opportunities during the auctions. A main feature of the designed institution is that it is a federation of auction houses, that is to say, different real fish markets can be connected to the institution allowing buyer agents to participate in several auctions at the same time. Thus, buyer agents receive information from several simultaneous auctions, being able to decide which is the most suitable place to buy. At the same time, the software in charge of the real auctions has been extended to permit the participation of software agents in the auction house. The connection between the real auction house and the electronic institution is done via sockets. On the one hand, the software of the real auction house sends to the electronic institution information of all the events occurring in the auction house relevant for the buyers. On the other hand, the electronic institution sends to the software in charge of the real auction house information about which buyer agents are taking part in the auction and the bids that they submit.

In what follows we concentrate on the virtual part concerning to the electronic institution which allows buyer agents to participate in the auction house federation. We want to point out that the auction house federation has been designed and developed making use of the software presented in this thesis. That is to say, the institution has been specified using the ISLANDER editor, and it is executed by loading the specification on the agents of the social layer. The staff agents playing the internal roles have been developed in JAVA.

7.1.1 Dialogic Framework

The dialogical framework defines all the roles that participating agents can play within the auction house federation. The institution contains the following roles:

Good Register (GR): it is an internal role which provides buyers with information about the goods registered in an auction house. That is, about the goods that will be auctioned later on. There is one agent playing the good register role for each real fish market connected to the federation.

Auction Broker (AB): it is an internal role which manages the scenes of a concrete auction. It informs agents about the events related to the auction occurring in the real fish market and also it sends the bids submitted by buyer agents to the software system in the real fish market. There is one agent playing the auction broker role for each real fish market connected to the federation.

DB Manager (DBM): it is an internal role which manages the database which contains historical information about the auction houses. It receives queries from the buyers, and it returns them the requested information,

if they are authorised to receive it. There is one agent playing the DB Manager role for the whole institution.

Buyer Admitter (bad): it is an internal role which controls buyer agents' access to the auction house federation. There is one agent playing the buyer admitter role for the whole institution.

Auction Admitter (aad): it is an internal role which controls buyer agents' access to a concrete fish market. There is one agent playing the auction admitter role for each real fish market connected to the federation.

Llotja: it is an internal role played by the agents connecting the auction house federation and a real fish market. Once in the institution they are responsible of the creation of all scenes related to a real fish market. There is one agent playing the role llotja for each real fish market connected to the federation.

Remote Control (RC): it is an internal role which performs all the tasks to participate in a specific auction under the control of a buyer agent. There is one agent playing the remote control role per buyer and auction.

Buyer Agent (BA): it is an external role played by the software agents whose aim is to buy goods in the auctions. Buyer agents receive information from several auction houses and can participate in each one thanks to a remote control agent. Each Buyer Agent coordinates the actions of several remote controls, each one taking part in a different auction. There is one buyer agent per user.

From the above description we can see that in the institution there are five internal roles and only one which is external. The external role belongs to the role *BuyerAgent* which should be customised with the user preferences before going to the institution to buy. Once in the institution, they should take into account their user preferences and the information that they receive from the different auction houses to decide which is the best place to buy. We want to point out that buyer agents do not participate directly in the auctions. They participate in the auctions through to an agent of the institution who takes the role of remote control and receives orders from the buyer and that participates in the auction on its behalf. A buyer agent has one remote control for each real auction in which it is taking part.

7.1.2 Performative Structure

Next we describe all the scenes in the auction house federation except the root and the output scenes, which are only used as the entry and exit points of the institution. In the auction house federation we find the following scenes:

Buyer Admission : this scene is devoted to control buyer agents access to the auction house federation. In this scene a *buyer admitter* is in charge

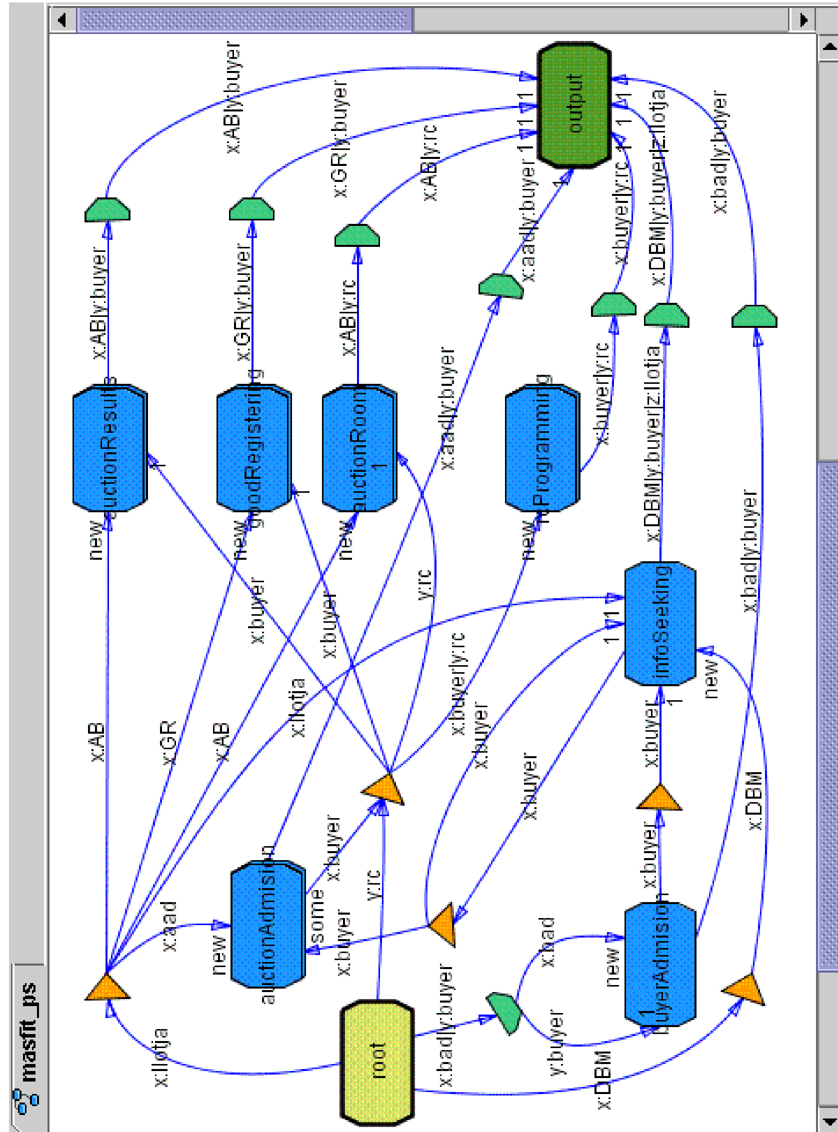


Figure 7.1: Graphical Specification of the aucton house federation performative structure.

of admitting buyer agents in the institution. Buyer agents entering in the scene request their admission by sending their username and password to the buyer admitter, who accepts them if their identification is correct; otherwise, it denies the admission. There is one buyer admission scene for the whole institution.

Auction Admission : this scene is devoted to control buyer agents access to the scenes of a concrete fish market and it is managed by the *auction admitter*. Whenever a buyer agent is admitted into a fish market, it is informed about which are the auction results and good registering scene of this fish market. There is one auction admission scene, managed by a different auction admitter, for each real fish market connected to the federation.

Auction Results : this scene is devoted to inform buyer agents about the results of the auctions. There is one auction results scene for each real fish market connected to the federation. In each one, the auction broker of an auction house and all the buyer agents currently taking part within that auction house, participate. The auction broker informs buyers about the result of each round. For each sale in the auction house, the auction broker informs the buyers about the identifier of the lot, the identifier of the buyer who has won the round, the price at which the buyer has won the round, and the number of acquired boxes. If a lot is withdrawn, buyers are informed about the identifier of the lot, the price at which the lot has been withdrawn, and the number of withdrawn boxes.

Good Registering : this scene is devoted to inform buyer agents about the lots registered within an auction house. There is one good registering scene for each real fish market connected to the federation. In each one the good register of an auction house and all the buyer agents currently taking part within that auction house participate. The good register informs the buyers of all the lots registered within the auction house. When a new buyer enters the scene it is informed about previously registered lots which have not been auctioned yet. Each registered lot is identified by the following attributes:

- an identifier for the lot;
- the fish species;
- the quality;
- the weight in kilograms;
- the ship that delivered the lot to the auction house;
- Wether the price during the auction will be per kilogram or per box;
- some characteristics about the presentation of the lot.

Auction Room : this scene is the most important, as it is the one in which the fish is auctioned. There is one auction room scene for each real fish market

connected to the federation. Each one is managed by an auction broker who auctions the lots following a descending bidding protocol. Besides an auction broker within the scene, one remote control for each buyer agent currently taking part within that auction house also participates. Each remote control participates in the auction room in behalf of a buyer agent from which it receives orders for submitting bids. In the next section we will explain in detail the auction room scene.

RC Programming : this scene is devoted to coordinate a buyer agent and a remote control. The buyer agent programmes the remote control by giving to it orders for buying within the auction room. These orders contain the identifier of the lot, the price to bid, and the number of boxes that the buyer wants. There is one rc programming scene for each buyer admitted in one of the real fish markets connected to the federation.

Info-Seeking : this scene is devoted to give buyer agents historical information about the different fish markets and about the fish markets connected at each moment to the federation. This scene is managed by the DB manager which receives requests from buyers and answers with the requested information, if buyers are authorised to have it. When the request is to know which are the real fish markets connected to the federation, the returned information by the DB manager contains the identifier of the admission scene for each fish market. That is, the scene into which the buyer agent must go, if it wants to participate in that fish market. Agents playing the role llotja can also participate in the scene. Concretely, when a fish market is connected to the federation, the agent playing the role llotja, which provokes the creation of the scenes for that fish market, moves to the info-seeking scene in order to inform the DB manager and buyer agents that a new fish market has connected to the federation.

Figure 7.1 depicts the graphical specification of the auction house federation performative structure. We can see, according to the description above, that there are seven scenes apart from the root and the output scenes. The figure also depicts the connections among the different scenes, which determines the paths that agents can follow depending on their role. Each scene, except RC programming scenes, is created by a staff agent playing one of the following internal roles: buyer admitter, auction admitter, auction broker, good register, and DB manager.

Notice that we differentiate between the admission into the auction house federation and the admission into a concrete fish market. The admission to the federation permits buyer agents to request historical information in the info-seeking scene but not to participate in the auctions. In order to participate in the auctions of an auction house connected to the federation buyer agents must first go to the corresponding auction admission scene. There are at least three reasons that justify to distinguish between the admission to the federation and the admission to a concrete auction house. Firstly, buyer agents may not be allowed to participate in all auction houses; for instance, a buyer agent may not

be registered in all the auction houses. Secondly, each auction house can have different admission rules. Lastly, each auction house wants to keep control over which buyer agents are authorised to participate and buy.

It has already been pointed out that each real fish market is connected to the auction houses federation via socket. There is one agent in charge of that socket that mediates the communication between the software in charge of the real fish market and the agents within the institution. This agent enters in the institution playing the role *llotja* and once in the root scene, it can only follow one path which provokes the creation of all the scenes for that auction house. Concretely, an execution of the following scenes is created: auction admission, auction room, auction results and good registering. Then, the agent in charge of the socket moves to all of them changing its role in each case. Concretely, it plays the auction admitter role in the auction admission scene, the auction broker role in the auction room and auction results scenes, and the good register role in the good registering scene. This agent receives information about all the events occurring in the real auction house which are relevant for the buyers through the socket. Depending on the type of the event, it informs the agents in the corresponding scene. For instance, when it receives a message informing about the registration of a new good, it informs all buyer agents within the good registering scene; and when it receives a message informing about the start of a round, it informs all remote controls within the auction room scene. Complementary, it informs the software in the real fish market whenever a new buyer is admitted within the auction house or when one of the participants leave, and about the bids submitted by the remote controls within the auction room scene. Notice that the decision about who wins each round is taken by the software in the real auction houses, not by the auction broker within the electronic institution.

Buyer agents entering in the institution must go first to the buyer admission scene where they ask for admission into the auction house federation. Once admitted they can go to the info-seeking scene where they can request historical information and information about which fish markets are connected to the federation. Then, from the info-seeking scene, they can move into the auction admission scenes of each fish market. Notice that the buyer can go to different auction admissions in parallel while at the same time, remain in the info-seeking scene. In this way buyer agents are informed about new fish markets connected to the federation and can try to enter in those fish markets in which it is not taking part. Furthermore, as they remain in the info-seeking scene, they can continue requesting information to the DB manager. When a buyer is admitted within an auction house a new remote control agent is launched. The buyer agent and its remote control have then to synchronise before moving to a newly created rc programming scene, and the rest of the scenes of that auction house. Concretely, the buyer agent goes to the auction results and to the good registering scenes of the auction house, and the remote control moves to the auction room scene in which it participates in behalf of the buyer agent. In the good registering scene, buyers receive information of the new lots as soon as they are registered in the

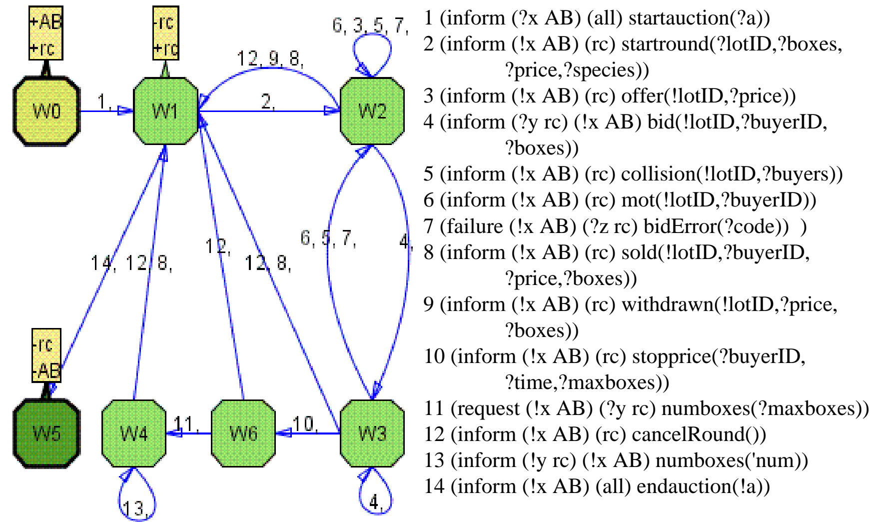


Figure 7.2: Specification of the auction room scene

real auction house. Thus, they have the same information as human buyers within the real fish market who can see the lots after being registered, and have some time to decide if they want to bid for the lot, and if so, at which price and the number of boxes that they want to acquire. This also gives buyer agents the necessary time to send the corresponding order to the remote control in the RC programming scene. Notice that buyer agents receive information from all the auction houses in which they have been admitted and then they can decide which is the best place to buy in. For each auction house they have a different remote control which participates in the auctions on its behalf. Using this approach, buyer agents can concentrate in the general buying strategy and they do not need to be directly controlling the evolution of each auctions in which they are taking part. For instance, if a buyer agent wants to submit a bid for the lot $l1$ at price 30 in one of the auction houses, it sends the order to the corresponding remote control, and does not need to be directly monitoring when the auction of lot $l1$ starts and when the offer reaches the value 30. Buyer agents are informed about the result of each round within the auction results scene.

7.1.3 Auction Scene

The main activity within a fish market is the auctioning of goods in the auction room. Figure 7.2 depicts the graphical specification of the auction room scene. In this scene can participate agents playing the auction broker and remote control roles. Concretely, the scene requires the participation of exactly one agent playing the auction broker role, while there is no limit on the number of remote controls which can participate within the scene. The graph depicts the states of the scene, along with the edges representing the legal transitions between scene states labelled with illocution schemes. The information contents of such schemes is expressed in prolog.

Notice that, apart from the initial and final states, the state $w1$ is labelled as an access and exit state for remote controls —meaning that between rounds some of the remote controls can leave and new ones might be admitted into the scene. A remote control only leaves the auction room scene, before it finishes, after receiving an order from the buyer agent which it is representing.

This scene is governed by the auction broker making use of the downward bidding protocol (DBP). The scene begins when the auction broker announces the start of the auction, label 1, and makes the scene evolve to $w1$. At $w1$ the auction broker can start a new round by sending the remote controls the information of the lot, label 2, or it can close the auction by moving the scene to the final state $w5$, label 5. The information of a lot sent to the remote controls consist on the identifier of the lot, the number of boxes, the initial price and the species of fish.

We have already pointed out that the auction brokers auctions the lot by a downward bidding protocol; once the round has been started all the buyers taking part in the round receive offers in descending value until there is a bid or the reserve price for the lot is reached and then the lot is withdrawn, or the round is cancelled. The auction house has always the capability to cancel the current round. If there is a bid the following situations can arise:

- "mot": each auction fixes an initial number of offers for which bids are not accepted. If a bid is submitted during this period then a "mot" is declared, and the round is restarted at a higher price. Human buyers use these first steps of the rounds to check that their electronic devices work properly without acquiring any good.

- collision: if there is more than one bid at the same price. Then, the round is restarted at a higher price.

- rejection of the bid: the auction house can always reject a bid submitted by a buyer. For instance when the buyer has not got enough credit. A bid error message is sent to the buyer who has submitted the bid and the round is restarted.

- sale: if there is only one bid and it is accepted. Then, the buyer who has submitted the bid is declared the winner of the round.

The state $w2$ represents the state where the auction broker sends the offers to remote controls. Notice that as there are also human buyers taking part in the auction a sale, a collision and a mot can be declared without any bid within the scene. These messages are consequence of bids submitted by human buyers, which are not depicted in the scene.

Bids submitted by remote controls contain the identifier of the lot, the identifier of the buyer whose remote control is representing, and the number of boxes that it wants acquire. The value at which the bid is submitted is the value of the last offer sent by the auction broker. If the value of the number of boxes in the winner bid is set to zero, the round is stopped, and the winner remote control is requested to inform how many boxes it wants to acquire. The decision about how many boxes to acquire is not taken by the remote control, it is taken by the buyer agent. Then, the remote control asks the buyer agent within the rc programming scene which decides how many boxes to buy by sending the corresponding message to the remote control.

7.2 Conference Centre

The second example corresponds to the Conference Centre (CC) [Arcos and Plaza, 2002] [2] A conference takes place in a physical setting, the conference centre, where different activities take place in different locations by people that adopt different roles (speaker, session chair, organisation staffer, etc.). During the conference people pursue their interests moving around the physical locations and engaging in different activities. In a moment in time people are physically distributed along the conference, possibly interacting with other people. We can easily think about the spatial proximity relations that exist among people in this physical space. However, if we think about an *informational space* where the past background and current interests of the conference attendees are represented, we could think of a new kind of *proximity* relation that is a function of the similarity among people's interests and backgrounds.

We can imagine software agents inhabiting the virtual space that take up some specific activities on behalf of the interest of an attendee of the conference. Specifically, a Personal Representative Agent (PRA) is an agent inhabiting the virtual space that is in charge of advancing some particular interest of a conference attendee by searching information and talking to other software agents.

Attendees have to instruct their PRAs specifying a presentation (e.g. a list of interested topics), an appearance (a collection of features describing the view an agent wants to offer to the other agents) for interacting with other PRAs, and a collection of tasks (e.g. meeting people, making appointments, etc) in which the PRA can participate for achieving the attendees' interests. The collection of tasks is provided by the Conference Centre definition as a set of scenes and roles in which a PRA can participate.

[2] We want to thank Josep Lluis Arcos for giving us information about the conference centre institution and a description of it, which we have used as the basis for this section.

Moreover, the Conference Centre provides two *mediation services* connecting the information space of agents and the physical space of human users: the *awareness service* and the *delivery service*.

The *awareness service* takes charge of pushing information from the physical space to the information space. Specifically, the awareness service provides to PRAs a real-time information about the physical location movements of users. The specific data provided depends on the particular sensors available in the awareness service for a particular application. For instance, in the conference centre application the awareness service provides a real-time tracking of attendees'location as well as the group of other attendees nearby a given attendee.

Concerning the *delivery service*, it offers mediation and brokerage capabilities (subscribed by the human users) for delivering information from the information space to the physical space. Specifically, the delivery service provides the channels for delivering the information gathered by the PRAs to their corresponding users. For instance, in the conference centre application the delivery service allows to send information as audio output by means of a wearable computer and HTML pages by means of screen terminals scattered through the conference building.

Apart from these two services connecting the two spaces, the Conference Centre provides a yellow pages service. PRAs can register in the yellow pages information about the attendee that they are representing and they can request information about other agents.

7.2.1 Dialogic framework

In the Conference Centre there are staff agents which give the conference centre services playing the internal roles *awarener* in charge of the awareness service, *deliverer* in charge of the delivery service and *broker* in charge of the yellow pages service.

Apart there are the *Personal Representative Agents* (PRAs) in charge of pursuing interests of attendees of the conference. Each attendee can launch several PRAs, each of them pursuing a different interest. During the conference a PRA can adopt different roles—and several at the same time if the PRA is participating in several scenes. The PRA roles are information gatherer, proposer, advertiser, information provider, context manager and information filterer. The roles that a PRA is playing depends on the tasks in which it is involved at each instant and on the responsabilities that its principal has —i.e. only PRAs belonging to a workshop organizer or to a demonstrator, or can adopt the role of advertisers.

PRAs have available information about the activities that take place in the Conference Centre and their scheduling. Examples of conference activities are exhibition booths and demo events, plenary and panel sessions, etc. They have also information about the different locations of the conference such as exhibition areas, conference rooms, and public areas—i.e. halls, cafeterias, and restaurants. This information is used by the agents to reason about the movements of users in the conference.

Furthermore, PRAs are customised by conference attendees via a WWW browser at registration to the conference. The attendee customises the PRA with the following information: i) an interest profile (specifying the topics the attendee is interested in); and ii) those tasks the user delegates the PRA to do in her behalf (e.g. if she is interested or not in making appointments).

7.2.2 Performative Structure

We can see in figure 7.3 the different scenes of the conference centre. It is important to remark here that, in order to perform these scenes, the information agents use *both* the information about the conference centre scheduling and locations, and the information received from the conference awareness service to infer the situation of the user. That is to say, knowing that the user is in a particular place, the current time, and the activity scheduled by the Conference for that place at that time, the information agent can infer the social activity in which the user is involved.

We will briefly summarize the tasks performed by PRAs and the scenes they are involved in except the root and exit scenes which only represent the enter and exit points of the institution.

Information Gathering Scene (IGS) : in this scene all the PRA's within the conference centre playing the information gatherer role participate together with a staff agent playing the broker role. The broker gives a yellow pages service to PRAs. Then, when a new PRA enters the scene it should register in the yellow pages by sending to the broker agent a message containing the name of the attendee whom it is representing, a list of topics in which she is interested and a list of information about those events in which she has some responsibility. Also PRAs can ask the broker agent for agents interested in a set of topics. We say that the information gathering scene constructs the *interest landscape* of a given attendee. The interest landscape holds all the information considered as useful for the interest of the attendee and is used and refined in the other tasks. When the information gathering task assesses a conference event with a high interest valuation, the information is directly delivered to the attendant via the delivery scene. In advertiser PRAs, this task has been specialized for attracting persons that might be interested in the conference events (exhibition booths or conference sessions) they represent.

Context Scene (CS) : in this scene, PRAs playing the context manager role, receive information from the conference awareness service for tracking the physical context of a given attendee. The conference awareness service keeps track of the whereabouts of the attendees in the Conference Centre. In the CC the detection devices are a network of infrared beacons (marking the different rooms, places and locations in the CC) and the wearable computers the attendees carry. The CC wearable computer detects the infrared beacons and thus informs the awareness service of the location of its user. Moreover, the wearable device possesses an infrared beacon,

Figure 7.3: Graphical specification of the conference centre performative structure.

allowing the detection of other persons, wearing wearable devices as well, located nearby. In order to have access to this information, each PRA in the information space "subscribes" its user to the awareness service. As a result, the PRA receives messages about the changes in location of that person and a list of other people close to that person. When an attendee is physically near another person, exhibition booth, or thematic session with similar interests to its, the PRA tries to inform the attendee via the delivery scene. Another task of the context scene is checking wether attendees are aware of their commitments (e.g. an appointment, but also commitments with the Conference organization, like chairing a session that is about to start, or boarding a bus that is about to leave for a tour the user has paid for). Commitments of attendees are only noticed when the context information available to PRAs indicates that the attendee is not aware of the commitment (e.g. it is five minutes before the starting of a session chaired by the attendee and the attendee is physically in a different place).

Appointment Negotiation Scenes (APS, ACS) : in these two scenes, and using the interest landscape, the PRAs try to arrange an appointment between two attendees. Both play the proposer role. The negotiation is done in two parts. First, PRAs negotiate a set of common topics for discussion (the meeting content) in the Appointment Proposal Scene (APS). The APS scene is explained in detail in the next subsection. If they reach an agreement about the topics, PRAs move to the Appointment Coordination Scene (ACS) where they negotiate about the appropriate meeting schedule.

Advertiser Scene (ADS) : in this scene a PRA, adopting the role of advertiser, tries to attract other PRAs, adopting the role of information filterer, to the conference event that the advertiser represents (workshop, booth, or demonstration). A PRA can only play the advertiser role if the attendee it is representing is responsible of an event.

Delivery Scene (DS) : is responsible of delivering information to the user by means of the conference delivery system. The delivery service in the CC allows the users to receive information in two ways: by means of a wearable computer with text and audio output and by screen terminals scattered through the Conference Centre. The wearable computer is used to convey short messages that are relevant for the user with respect to her current physical and social surroundings. The user can walk to a terminal if she wishes to have more information about this message or other recent messages she has received. When the user approaches a screen the wearable computer detects this terminal's identifier, and then it sends this identifier to the user's PRA. Once the PRA is aware of this situation, the agent sends to that screen the report of the performed tasks and the report of ongoing tasks.

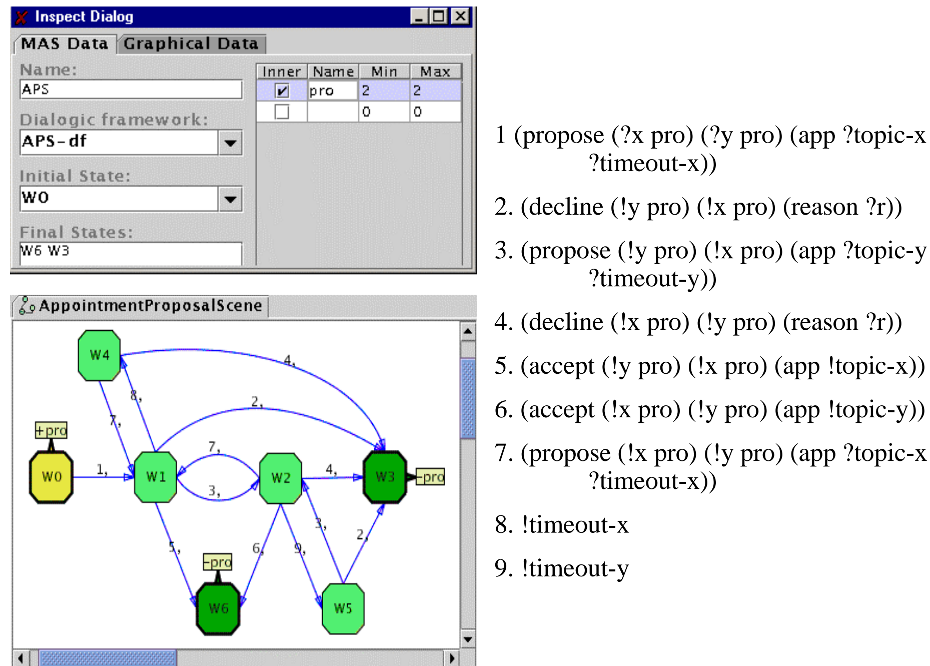


Figure 7.4: Specification of the Appointment Proposal Scene.

7.2.3 Appointment Proposal Scene

Figure 7.4 depicts the specification of the Appointment Proposal Scene. The participants of this scene are two personal representative agents PRA_x and PRA_y playing the role of proposer. The goal of the scene is to agree upon a set of topics for discussing in the appointment—represented in Figure 7.4 as `(app ?topic-x ?timeout-x)`, `(app ?topic-y ?timeout-y)`, where the arguments of `app` are the set of topics and the caducity of the proposal. Following the interaction protocol shown in Figure 7.4, the scene is played as follows:

1. one of the PRAs takes the initiative and sends an appointment proposal to the other PRA, label 1, with a set of initial topics `topic-x` and a timeout `timeout-x` defining a caducity for the proposal. We will refer to the initiating agent PRA_x and to the other PRA_y. This set of topics is intended to be a subset of the attendant's posted profile of interests.

2. PRA_y evaluates the proposal ($w1$) and can either (i) accept (transition to $w6$), (ii) decline (transition to $w3$), or (iii) send a counter proposal to PRA_x with a (partially) different set of topics and a new timeout (transition to $w2$). Whether the `timeout-x` expires and PRA_y has not answered, the

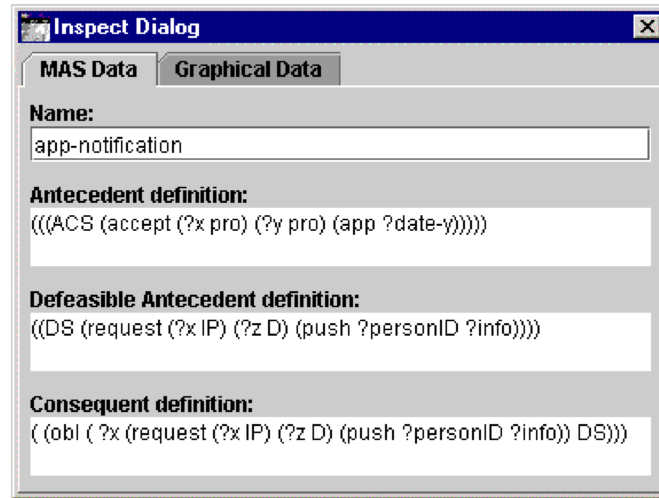


Figure 7.5: Specification of a norm in the conference centre.

scene moves to $w4$.

3. in turn, when PRA_x receives the counter proposal of PRA_y evaluates it and can also either accept (transition to $w6$), decline (transition to $w4$), or send a counter proposal to PRA_y (transition to $w1$). This negotiation phase finish when an agreement on topics is reached or one of them decides to withdraw it.

4. when `timeout-x` expires without any answer from PRA_y (state $w4$), PRA_x can either send a new proposal (transition to $w1$) or finish the scene with a decline message (transition to $w3$).

5. when `timeout-y` expires without any answer from PRA_x (state $w5$), PRA_y can either send a new proposal (transition to $w2$) or finish the scene with a decline message (transition to $w3$).

When PRAs reach the acceptance final state, the scene finishes but remark that, up to this point, no time commitment was made—thus no change in the agenda of attendants have yet taken place. Nevertheless, the acceptance of appointment contents in APS scene involves the commitment of accepting the appointment if PRAs reach a time-slot agreement in the ACS scene. We adopt this lazy commitment strategy in order to facilitate the management of the agenda.

7.2.4 Norms

As we have said, PRAs participate in the virtual space representing an attendee and looking for interesting activities and appointments for the user. In order to send information to the user the institution gives PRAs a delivery service. Thus, when a PRA reaches an agreement for an appointment with another PRA, or it decides that an event is interesting for its user, the PRA must inform the attendee about it. Figure 7.5 depicts the norm which obliges PRAs accepting another PRA proposal for an appointment date, in the ACS scene, to inform the user about the appointment within the delivery scene. Remember, that appointment negotiations are done in two steps, first PRAs negotiate the appointment topics in the APS scene, and if they reach an agreement about the topics, they negotiate about an appointment date in the ACS scene. So, when a PRA accepts another PRA proposal for a date for an appointment, it implies that they have previously reached an agreement about the appointment topics. A similar norm obliges the other PRA, that is, the one which receives the accept message, to go to the DS scene and inform its user. Notice that although the ACS scene is connected to the output scene this path is disabled for those PRAs which have the obligation to inform their users about the appointment. That is, the path that connects the ACS scene and the output scene can only be followed by those PRAs which have not reached an agreement for an appointment. Another norm in the conference centre obliges PRA's which accept another PRA proposal to attend an event in the AD scene, to go to the DS scene and inform their user.

7.3 Conclusions

In this chapter we have presented two examples of institutions: the auction house federation and the conference centre. The auction house federation is an institution which permits buyer software agents to participate in real fish markets in the same conditions as human buyers. For this purpose, the institution can be connected to the software at each auction house from which it receives information about the events occurring within it. Thus, buyer agents within the federation receive information from several auction houses and they decide which are the best place to buy. We want to remark that the design and development of the auction house federation have been done using the software tools presented in this thesis. Firstly, the institution has been specified and verified using the ISLANDER editor. Once the institution has been verified it is executed using the infrastructure architecture proposed in chapter 6.

The second example correspond to the conference centre institution where Personal Representative Agents interact in order to look for interesting events for the attendee they are representing. PRAs are customised before to go to the institution by the user preferences. The institution provides PRAs with the necessary services to receive and send information to the physical world.

Chapter 8

Conclusions

Due to the expansion of the Internet and the facilities that it offers to communicating entities, open multi agents systems have become the most promising application area of multi agent systems [Wooldridge et al., 2000]. Thus, the design and development of open multi-agent systems has become a fundamental issue in agent research. The complexity of this type of systems requires for appropriate methodologies and software tools that guide and give support to their design and development [Iglesias et al., 1999, Jennings et al., 1998]. In this thesis we have focused on the design and development of open multi-agent systems: systems whose components are unknown in advance, can change over time, and are composed by heterogeneous (human and software) agents probably developed by different people using different architectures and languages.

We have argued that open multi agent systems can be designed and developed as agent mediated electronic institutions. Electronic institutions define the rules of the game within agent societies, likewise institutions do in human societies. Institutions define what agents are permitted and forbidden to do, and more importantly they are in charge of enforcing their rules. Then, following the work of [Noriega, 1997, Rodríguez-Aguilar, 2001], we have concentrated on formalising and providing support to the specification and development of electronic institutions.

Because of the complexity of electronic institutions we have opted for a formal approach in their design and development. Then, in chapter 3, we have presented a formalisation of electronic institutions, following the work in [Noriega, 1997, Rodríguez-Aguilar, 2001]. Thus, we have described and presented a formal definition of the components of an institution. The institution formalisation represents a sound basis on which to guide the design and development of institutions. Based on the institution formalisation we have defined a textual specification language called ISLANDER. We believe that the process of specifying an institution forces the designer to go through a deep analysis of the problem and allows him to detect critical parts of the system before starting its development. We have proposed that agent engineers intending to design electronic institutions specify their components, namely:

- The *dialogic framework*, which includes the definition of a common ontology that allows agents to understand each other, the selection of a content language used to express the knowledge in communication language expressions, and the definition of set of internal and external roles along with the relationships among them.
- The *scenes*. Each scene defines a conversation protocol for a group of roles. Each scene requires the definition of its participating roles and their population, its conversation protocol, and the states at which agents can either leave or join the conversation. The scenes of an institution define the valid interactions that agents may have and they are the context wherein exchanged illocutions among agents must be interpreted.
- The *performative structure* capturing the relationships among the different scenes. The performative structure defines how agents depending on their role can move among the different scenes, whether they have to synchronise with other agents or not, and whether new scene executions are created.
- The set of *norms* which capture the consequences of agents' actions within the different scenes.

In order to give support to institution designers through the specification process, we have built an ISLANDER editor. The tool combines textual and graphical specification of the institution components. We believe that graphical specifications are extremely useful, since they facilitate the designer work and the understanding of the specification by other people. Furthermore, the tool permits the verification of institution specifications. This is an essential step before to start the development of the system or in our case before loading the specification in the agents composing the institution infrastructure. We have to take into account that debugging and finding errors in distributed systems is a difficult and hard task. From this arise that errors detected at specification stage may save a lot of time and effort to system developers.

An important feature is that specifications done in ISLANDER are independent of any programming language, and so they do not impose restrictions on which language has to be used to develop an infrastructure for the institution and the agents taking part in it. Furthermore, the outputs generated by the tool can be translated into different languages and be used for the development of infrastructures and agents in any language. For instance, in section 6.4 specified institutions in ISLANDER are translated into a logical formalism expressed in Prolog. Besides, the notion of scene and performative structure are general enough to be used for specifying agent interactions and activities out of the scope of an institution. For instance, in [Ontañón and Plaza, 2002] ISLANDER has been used to define the interaction protocol among a group of learning agents.

Complementarily, in chapter 4 we have focused on the formalisation of multi agent systems with process algebras. Process algebras focus on the formalisation and analysis of distributed and concurrent systems. Then, we believe that they can be a useful mechanism to analyse and verify multi-agent systems. In this

chapter, we have presented an alternative architecture for auction systems, by eliminating the auctioneer and proposing a distributed bidding resolution mechanism. Concretly we have adapted a well known distributed algorithm as the leader election algorithm to determine the winner at each round. The system has been specified in $\pi-$calculus a type of process algebra.

We advocate that the execution of an institution requires an infrastructure which facilitates agents participation within the institution while enforcing institutional rules. As we do not impose restrictions on the agents which can participate within an institution, we can not expect that those agents will behave according to the institutional rules. On the one hand, the infrastructure provides participating agents with the information they need to successfully participate in the institution, and facilitates their interaction with the rest of the agents. On the other hand, the infrastructure is in charge of enforcing the institution rules to participating agents as well as of having control of each agent pending obligations. In other words, the infrastructure only allows agents to perform those actions which are correct with respect to the institutional rules. In order to achieve these objectives we have developed an agent-based social layer middleware between the communication layer (implemented by the JADE platform [Bellifemine et al., 2001]) and the agent layer. An important feature of the developed social layer is that it can be used in the deployment of different institutions because the agents composing it are capable of loading institution specifications as generated by the ISLANDER editor. The agents composing the social layer coordinate to guarantee the correct evolution of scenes, to control the agents' movements among scenes, and to control the obligations of each agent. The most important agent of the social layer is the governor, a special type of mediator agent devoted to mediate between an agent and the environment in which it is situated. Each agent within the institution is connected to a governor which mediates its communication with the rest of the agents within the institution, verifies that its behaviour is correct with respect to the institutional rules and keeps track of the agent pending obligations. Notice that the social layer also protects agents from other agents' fraudulent behaviour. For instance, the social layer prevents agents of being overloaded by an agent which is constantly sending incorrect messages to other agents.

In order to illustrate how electronic institutions can be specified we have presented two examples in chapter 7: the auction house federation and the conference centre. The auction house federation is an institution devoted to permit buyer software agents to participate in real fish markets in the same conditions as human buyers. Concretely, several real fish markets can be connected to the institution permitting buyer agents to participate in several auctions at the same time. Buyer agents receive information from all of them and they decide which are the better places to buy. Notice that the resulting specification must be regarded as a multi-market institution, differently to the single market institution presented in [Noriega, 1997, Rodríguez-Aguilar, 2001]. The second example corresponds to a conference centre. Besides a physical space in which a conference takes place, a virtual space populated by agents representing conference

attendees is defined. The virtual space is populated by Personal Representative Agents(PRA) each one customised with an attendee preferences. Then, PRAs within the conference centre institution devote their time interacting with other agents in order to look for interesting activities for the attendee they represent.

Finally, we want to summarise the steps that from our point of view should be followed in order to design and develop nstitutions:

- Specification of an institution.
- Verification of the specification.
- Development of the staff agents.

Obviously we advocate that the ISLANDER editor should be used for the specification and verification of institutions. Once the institution has been specified and verified, the staff agents should be developed. Since institutions delegate their services to staff agents their development is necessary before making the institution accessible to external agents. Currently, partial support is provided to agent developers. Along this direction, in section 6.4 we have shown how agent skeletons can be synthesised from the institution specification and then, customised by agent designers.

After going through these steps an institution can be executed by launching the institution infrastructure composed of the JADE platform, the social layer, and the staff agents. Thereafter, external agents can be admitted to participate within the institution. Notice that using this approach institution designers do not have to spend time on infrastructure development. They only must inform the social layer agents about where the institution specification is stored. We believe that this represents an important step forward as it reduces the amount of work and time necessary to develop an institution. Thus, institution designers can devote their time to domain dependent issues related to the institution that they aim at developing. In other words, their workload is narrowed down to the specification of an institution along with the development of the staff agents to which the institution delegates its services.

8.1 Future Work

We believe that this thesis represents an important advance on the design and development of open multi-agent systems. Nonetheless, there is an important amount of work to do in the area.

Firstly, there are some extensions of the institution formalisation and specification language, which we think can be useful and which would cover some limitations of our approach. These extensions relate to the following issues:

Role relationships . Our model permits the specification of role relationships but there is no way to specify which restrictions they impose on the interactions and activities that agents playing the roles can do within an institution. Currently, it is a task of the institution designer to take this

into account when it specifies the rest of the components of an institution. We think that at a first step users should specify the restrictions that each role relationships imposes. Then, it should be authomaticaly verified that the designed institution does not violate these restrictions.

Execution Information . As pointed out in chapter 6, an issue to address is which information about an institution execution is given to the agents. That is to say, information about the different scene executions and about the participants in each of them. This information can be useful as it can permit agents to detect interesting scenes to join. On the contrary, in some cases it would not be desirable to give agents all the information about the institution execution. That is, there is some information that should be kept private by the institution. For instance, the institution may keep private that two agents are involved in a negotiation scene. We believe that it is a task of the institution to define to which execution information agents have access and it should be associated to roles. That is to say, different roles may have access to different information. Therefore, the institution designer should be permitted to define which information about the institution execution is authorised to acces to each role.

Role attributes . In the current formalisation a role defines a pattern of behaviour within an institution. That is, at each moment the role that an agent is playing constraints what the agent can do. We believe that this could be extended associating to each role some attributes. For instance, a buyer within an auction house may have associated a credit and a list of purchased goods. Apart from defining the role attributes, it must be defined when and how the value of each attribute can change. For instance, when a buyer wins a round, its credit will be decremented. Then, the values of the attributes can be used to determine the actions that an agent can do. That is to say, they can be used in scene and performative structure constraints. For instance, a buyer can not be allowed to submit bids greater than its credit. Taking this approach, roles can be seen as abstract data types whose definition contain the role attributes and how and when attributes values have to be modified.

Norms . Currently only dialogic actions can appear in norm definitions. Then, it is not possible to express that an agent is obliged to move to a scene, or that it has to synchronise with an agent or a group of agents in order to start a new scene execution. We believe that norm definitions should be extended to cope with these possibilities. Another issue to address related to norms is when an agent has to fulfil their obligations. Currently, the meaning of norms is that agents have to fulfil their obligations at some instant in the future, before leaving the institution. This is satisfactory on some of the cases but not in all of them. We believe that norm definitions should permit to express *when* an agent has to fulfil their obligations. For instance, that the agent has to fulfil an obligation immediately, at the first

opportunity that it has, or define a period of time that it has to fulfil the obligations.

Notice that these extensions will imply to extend the ISLANDER editor with new verification capabilities. Furthermore, social layer agents will have to be extended to cope with these extensions. Also, as we pointed out in chapter 5, more properties must be verified with respect to scene constraints and conditions on performative structure arcs. Currently, scene and performative structure constraints are not taken into account when verifying that a scene can not be blocked at any non final state and that agents will not be blocked within the institution performative structure.

Scene constraints capture the restrictions that the previous interaction within a scene imposes on its future evolution. Constraints restrict the valid values for illocution scheme variables and the paths that the scene evolution can take. It should be verified that a scene evolution cannot reache a state in which there is no illocution that agents can utter that satisfies the arcs' constraints. In order to do so, all possible dialogues that agents may have, the value of each scene variable and the different scene constraints must be taken into account. We believe that model checking or CSP can be used in this process.

The performative structure defines how agents can move among different scenes. Concretely, each arc label determines the roles that can progress through it. The outgoing arcs of a scene determine the paths that agents within the scene can follow when leaving it. Furthermore, each outgoing arc can have some associated conditions that agents must satisfy in order to progress through the arc. These conditions are expressed as illocution schemes and obligations. It should be verified that an agent can not be blocked within an scene because it does not satisfy any of the conditions associated to the scene outgoing arcs labelled with its role and no illocution uttered within the scene can change this situation. This condition should be verified for each scene and for each role that can be played within it. At this aim, all the different paths and scenes that an agent can follow from the initial scene should be analysed.

We have developed several software components which give support to the design and execution of electronic institutions. Concretely, we have developed a specification and verification tool, and a generic infrastructure. Nonetheless, we believe that this work should continue as more tools are needed. As a next step we advocate for developing tools which give support to agent development and institution monitoring.

We believe that the first point to address is to give support to agent developers. This is specially important in the case of staff agents because these are the agents to which the institution delegates its duties and services, and they are absolutely necessary for the execution of an institution. The first step should be the authomatic generation of agent skeletons or templates in different languages. As a result, library of agent skeletons per role could be given to agent designers. Then, we propose the development of an agent builder application which would permit agent designers to complete their agents by selecting first the scenes in which its agent will be involved, and defining later on the behaviour within each

scene. That is to say, the designer should fill up the skeletons with the decision making procedures. The work reported in section 6.4 for the synthesis and customisation of agents in Prolog, it is a first step in this direction. The work done in the ISLANDER editor can also be very useful in the development of an agent builder as it would permit to present a graphical representation of the institution to agent designers.

The agent builder should give support to agent development in different languages. Concretely, we think of giving support to agent development in general-purpose programming languages (such as Prolog, Lisp and Java) and higher-level, agent-oriented programming languages (such as 3APL [3APL, URL]). Notice that some of the user decisions are language independent. Although the main motivation for the development of an agent builder is to give support to the development of the staff agents, we believe that the tool would also be very useful for the designers of external agents.

The purpose of the monitoring tool is to reproduce what has happened during the execution of an institution, in a similar way as the monitoring tool developed for the fish market electronic auction house did [Rodríguez-Aguilar, 2001]. We believe that the monitoring tool should permit to reproduce institution executions at three levels:

- Performative structure: showing which scenes are executed within the institution at each instant, along with their participants. The tool should show at this level when new scene executions are created and when they finish, as well as the agent movements among them.
- Scene: showing the execution of a concrete scene. That is, the participants within a scene, the scene state and the messages that agents within the scene exchange.
- Agent: showing the institution execution from the point of view of an agent. That is, in which scenes is the agent involved, what messages it sends and receives in each one, and what obligations has it adquired.

In order to reproduce an institution execution an agents composing the social layer must store information about all the events produced during the execution. That is to say, about the creation and finalisation of scene executions, about agent movements among the different scenes, about all the illocutions uttered within the different scenes, and about the obligations acquired and fulfilled by each one of the agents. Then, the monitoring tool would use this information and the institution specification to reproduce institution execution. The tool should permit users to customise at each level which information they wants to see.

Trust and accountability are the main motivations for the development of a monitoring tool for institutions [Noriega, 1997, Rodríguez-Aguilar, 2001]. Giving accountability information to the participants will increase they trust in the institution. This is specially important for electronic institutions where people delegate their tasks to agents. Furthermore, it would permit them to analyse

their agent(s) behaviour within the institution and improve them. From the point of view of the institution designers, the tool could serve to test the system and the staff agents before making the institution available to external agents. Furthermore, when the institution is running it can be used to detect unexpected situations and fraudulent behaviours of external agents.

We believe that the simulation and development of test-beds is another important issues to address. The simulation of institutions will permit to study their performance. For this purpose, the simulation platform presented in [Vasconcelos et al., 2003] is an initial step in that direction.

The last issue that we want to point out as future work is institution evolution. Electronic institutions are situated in a dynamic environment which change over time, then, institutions should be capable to adapt to those changes. This has been identified as a main feature of open multi-agent systems [Jennings, 2001]. The capability of evolving is an important feature of human institutions, but this is not possible in our model. That is to say, the structure of the institution is static and can not evolve at execution time. It should be studied how an institution can evolve. That is say, *which* elements and *how* can be changed at execution time. We believe that one way in which this could be achieved is by adding a meta level to our model where agents interaction will be about the institution structure and its performance.

Appendix A

ISLANDER specifications

In this appendix we provide the textual specification of the electronic auction house federation and the conference centre.

A.1 Auction house federation

```
(define-institution masfit_institution as
  dialogic-framework = masfit-df
  performative-structure = masfit-ps
)

(define-performative-structure masfit-ps as
  scenes = (
    (rcProgramming rc-programming-scene list)
    (goodRegistering good-registering-scene list)
    (auctionResults auction-results-scene list)
    (infoSeeking info-seeking-scene)
    (root root-scene)
    (buyerAdmision buyer-admision-scene)
    (auctionRoom auction-room-scene list)
    (output output-scene)
    (auctionAdmision auction-admision-scene list))
  transitions = (
     (createInfoSeeking AND)
     (toBuyerAdmision OR)
     (toRCProgRC AND)
     (toAuctionAdmision AND)
     (exitInfoSeeking OR)
     (exitGoodRegistering OR)
     (createLlotja AND)
     (exitAuctionResults OR)
```

```
    (exitRCProgramming OR)
    (exitAuctionAdmision OR)
    (exitBuyerAdmision OR)
    (exitAuctionRoom OR)
    (toInfoSeeking AND))
  connections = (
    (infoSeeking toAuctionAdmision (((x buyer))))
    (auctionResults exitAuctionResults (((x AB))((y buyer))))
    (auctionAdmision exitAuctionAdmision (((x aad))((y buyer))))
    (root toRCProgRC (((y rc))))
    (buyerAdmision toInfoSeeking (((x buyer))))
    (buyerAdmision exitBuyerAdmision (((x bad))((y buyer))))
    (goodRegistering exitGoodRegistering (((x GR))((y buyer))))
    (root toBuyerAdmision (((x bad))((y buyer))))
    (rcProgramming exitRCProgramming (((x buyer))((y rc))))
    (auctionRoom exitAuctionRoom(((x AB))((y rc))))
    (root createInfoSeeking (((x DBM))))
    (root createLlotja (((x llotja))))
    (infoSeeking exitInfoSeeking (((x DBM))((y buyer))((z llotja))))
    (auctionAdmision toRCProgRC (((x buyer))))
    (toRCProgRC auctionRoom(((y rc))) 1)
    (exitRCProgramming output(((x buyer)(y rc))) 1)
    (exitAuctionResults output(((x AB))((y buyer))) 1)
    (createInfoSeeking infoSeeking(((x DBM))) new)
    (exitAuctionRoom output(((x AB))((y rc))) 1)
    (createLlotja auctionAdmision (((x aad))) new)
    (exitAuctionAdmision output(((x aad))((y buyer))) 1)
    (exitInfoSeeking output(((x DBM))((y buyer))((z llotja))) 1)
    (toRCProgRC auctionResults(((x buyer))) 1)
    (toAuctionAdmision auctionAdmision(((x buyer))) some)
    (exitGoodRegistering output(((x GR))((y buyer))) 1)
    (createLlotja goodRegistering(((x GR))) new)
    (toAuctionAdmision infoSeeking(((x buyer))) 1)
    (createLlotja infoSeeking(((x llotja))) 1)
    (toRCProgRC goodRegistering(((x buyer))) 1)
    (createLlotja auctionResults(((x AB))) new)
    (toInfoSeeking infoSeeking(((x buyer))) 1)
    (toRCProgRC rcProgramming(((x buyer))((y rc))) new)
    (toBuyerAdmision buyerAdmision(((y buyer))) 1)
    (toBuyerAdmision buyerAdmision(((x bad))) new)
    (createLlotja auctionRoom(((x AB))) new)
    (exitBuyerAdmision output(((x bad))((y buyer))) 1))
   initial-scene = root
   final-scene = output
)
```

```
(define-scene info-seeking-scene as
  roles = (llotja DBM buyer)
  scene-dialogic-framework = info-seeking-df
  states = (W1 W3 W0 W2)
  initial-state = W0
  final-states = (W3)
  access-states = ((llotja (W1 W0)) (DBM (W0)) (buyer (W1 W0)) )
  exit-states = ((llotja (W1 W3)) (DBM (W3)) (buyer (W1 W3)) )
  agents-per-role = (
       (1 <= DBM <= 1))
  connections = (
     (W1 W1 (inform (?z llotja) (all) registerLlotja(Infollotja(?name,
               ?admissionID)))  )
     (W1 W2 (request (?y buyer) (!x DBM) query(?sql))  )
     (W2 W1 (inform (!x DBM) (!y buyer) llotges(?info_llotges))  )
     (W2 W1 (failure (!x DBM) (!y buyer) error(?codeerror))  )
     (W1 W3 (inform (!x DBM) (all) close())  )
     (W0 W1 (inform (?x DBM) (all) open())  )
     (W1 W2 (request (?y buyer) (!x DBM) activeLlotges())  )
     (W2 W1 (inform (!x DBM) (!y buyer) result(?info))  )
  )
)

(define-scene auction-admision-scene as
  roles = (buyer aad)
  scene-dialogic-framework = auction-admision-df
  states = (W3 W0 W1 W2)
  initial-state = W0
  final-states = (W3)
  access-states = ((buyer (W0 W1)) (aad (W0)) )
  exit-states = ((buyer (W3 W1 W2)) (aad (W3 W2)) )
  agents-per-role = (
     (buyer <= 1)
     (1 <= aad <= 1))
  connections = (
     (W0 W1 (inform (?x aad) (all) open())  )
     (W2 W1 (failure (!x aad) (!y buyer) deny(?code))  )
     (W2 W1 (inform (!x aad) (!y buyer) acceptauction(?idAuctionResults,
              ?idGoodRegistration))  )
     (W1 W2 (request (?y buyer) (!x aad) loginauction(?username,
               ?password))  )
     (W1 W3 (inform (!y aad) (all) close())  )
  )
```

```
)

(define-scene auction-room-scene as
  roles = (rc AB)
  scene-dialogic-framework = auction-room-df
  states = (W5 W6 W0 W3 W2 W4 W1)
  initial-state = W0
  final-states = (W5)
  access-states = ((rc (W0 W1)) (AB (W0)) )
  exit-states = ((rc (W5 W1)) (AB (W5)) )
  agents-per-role = (
      (1 <= AB <= 1))
  connections = (
      (W1 W5 (inform (!x AB) (all) endauction(!a))  )
      (W3 W2 (inform (!x AB) (rc) mot(!lotID,?buyerID))  )
      (W2 W1 (inform (!x AB) (rc) sold(!lotID,?buyerID,?price,?boxes))  )
      (W2 W1 (inform (!x AB) (rc) withdrawn(!lotID,?price,?boxes))  )
      (W2 W2 (inform (!x AB) (rc) mot(!lotID,?buyerID))  )
      (W4 W4 (inform (!y rc) (!x AB) numboxes(?num))  )
      (W3 W2 (inform (!x AB) (rc) collisio(!lotID,?buyers))  )
      (W3 W3 (inform (?y rc) (!x AB) bid(!lot,?buyer_ID,?boxes))  )
      (W0 W1 (inform (?x AB) (all) startauction(?a))  )
      (W4 W1 (inform (!x AB) (rc) sold(!lotID,?buyerID,?price,?boxes))  )
      (W2 W2 (inform (!x AB) (rc) offer(!lotID,?price))  )
      (W2 W1 (inform (!x AB) (rc) cancelRound())  )
      (W2 W2 (inform (!x AB) (rc) collision(!lotID,?buyers))  )
      (W3 W1 (inform (!x AB) (rc) sold(!lotID,?buyerID,?price,?boxes))  )
      (W3 W2 (failure (!x AB) (?z rc) bidError(?code))  )
      (W2 W2 (failure (!x AB) (?z rc) bidError(?code))  )
      (W6 W1 (inform (!x AB) (rc) cancelRound())  )
      (W1 W2 (inform (!x AB) (rc) startround(?lotID,?boxes,?price,?species))  )
      (W3 W6 (inform (!x AB) (rc) stopprice(?buyerID,?time,?maxboxes))  )
      (W6 W4 (request (!x AB) (?y rc) numboxes(?maxboxes))  )
      (W3 W1 (inform (!x AB) (rc) cancelRound())  )
      (W4 W1 (inform (!x AB) (rc) cancelRound())  )
      (W2 W3 (inform (?y rc) (!x AB) bid(!lotID,?buyerID,?boxes))  )
  )
)

(define-scene buyer-admision-scene as
  roles = (buyer bad)
  scene-dialogic-framework = buyer-admision-df
  states = (W3 W0 W2 W1)
  initial-state = W0
  final-states = (W3)
```

```
  access-states = ((buyer (W0 W1)) (bad (W0)) )
  exit-states = ((buyer (W3 W1)) (bad (W3)) )
  agents-per-role = (
      (buyer <= 1)
      (1 <= bad <= 1))
  connections = (
      (W0 W1 (inform (?x bad) (all) open())  )
      (W2 W1 (inform (!x bad) (!y buyer) accept())  )
      (W2 W1 (failure (!x bad) (!y buyer) deny(?code))  )
      (W1 W3 (inform (!x bad) (all) close())  )
      (W1 W2 (request (?y buyer) (!x bad) login(?username,?password))  )
  )
)

(define-scene auction-results-scene as
  roles = (AB buyer)
  scene-dialogic-framework = auction-results-df
  states = (W2 W1 W0)
  initial-state = W0
  final-states = (W2)
  access-states = ((AB (W0)) (buyer (W1 W0)) )
  exit-states = ((AB (W2)) (buyer (W2 W1)) )
  agents-per-role = (
      (1 <= AB <= 1))
  connections = (
      (W0 W1 (inform (?x AB) (all) open())  )
      (W1 W1 (inform (!x AB) (buyer) sold(?lot,?buyer_ID,?price,?boxes))  )
      (W1 W2 (inform (!x AB) (all) pwd)  )
      (W1 W1 (inform (!x AB) (buyer) withdrawn(!lotID,?price,?boxes))  )
  )
)

(define-scene good-registering-scene as
  roles = (GR buyer)
  scene-dialogic-framework = good-registering-df
  states = (W2 W0 W1)
  initial-state = W0
  final-states = (W2)
  access-states = ((GR (W0)) (buyer (W0 W1)) )
  exit-states = ((GR (W2)) (buyer (W2 W1)) )
  agents-per-role = (
      (1 <= GR <= 1))
  connections = (
      (W0 W1 (inform (?x GR) (all) open())  )
      (W1 W1 (inform (!x GR) (?y buyer) oldgood(?lot,?especie,?quality,
```

```
                ?boxes,?kg,?price_type,?ship,?presentation))  )
      (W1 W2 (inform (!x GR) (all) close())  )
      (W1 W1 (inform (!x GR) (buyer) newgood(?lot,?especie,?quality,
               ?boxes,?kg,?price_type,?ship,?presentation))  )
  )
)

(define-scene root-scene as
  roles = (bad DBM buyer rc llotja)
  scene-dialogic-framework = masfit-df
  states = (W0)
  initial-state = W0
  final-states = (W0)
  access-states = ((bad (W0)) (DBM (W0)) (buyer (W0)) (rc (W0)) (llotja (W0)) )
  exit-states = ((bad (W0)) (DBM (W0)) (buyer (W0)) (rc (W0)) (llotja (W0)) )
  connections = (
  )
)

(define-scene output-scene as
  roles = (llotja GR bad DBM buyer aad AB rc)
  scene-dialogic-framework = masfit-df
  states = (W0)
  initial-state = W0
  final-states = (W0)
  access-states = ((llotja (W0)) (GR (W0)) (bad (W0)) (DBM (W0)) (buyer (W0))
      (aad (W0)) (AB (W0)) (rc (W0)) )
  exit-states = ((llotja (W0)) (GR (W0)) (bad (W0)) (DBM (W0)) (buyer (W0))
      (aad (W0)) (AB (W0)) (rc (W0)) )
  connections = (
  )
)

(define-scene rc-programming-scene as
  roles = (rc buyer)
  scene-dialogic-framework = rc-programming-df
  states = (W2 W3 W0 W1)
  initial-state = W0
  final-states = (W3)
  access-states = ((rc (W0)) (buyer (W0)) )
  exit-states = ((rc (W3)) (buyer (W3)) )
  agents-per-role = (
       (1 <= rc <= 1)
       (1 <= buyer <= 1))
  connections = (
```

```
      (W1 W3 (inform (!y buyer) (!x rc) close())  )
      (W1 W3 (inform (!x rc) (!y buyer) close())  )
      (W1 W1 (request (!y buyer) (!x rc) programbid(?lotID,?price,?boxes))  )
      (W0 W1 (inform (?x rc) (?y buyer) open())  )
      (W1 W2 (inform (!x rc) (!y buyer) priceStopped(?idLot,?time,?maxboxes))  )
      (W2 W1 (inform (!y buyer) (!x rc) numberBoxes(!idLot,?boxes))  )
  )
)

(define-dialogic-framework buyer-admision-df as
  ontology = buyer-admission-ontology
  content-language = PROLOG
  illocutionary-particles = (inform failure request )
  external-roles = (buyer )
  internal-roles = (bad )
  social-structure = ()
)

(define-dialogic-framework good-registering-df as
  ontology = good-registering-ontology
  content-language = PROLOG
  illocutionary-particles = (inform )
  external-roles = (buyer )
  internal-roles = (GR )
  social-structure = ()
)

(define-dialogic-framework masfit-df as
  ontology = masfit-ontology
  content-language = PROLOG
  illocutionary-particles = (request inform failure )
  external-roles = (buyer )
  internal-roles = (GR bad DBM AB aad llotja rc )
  social-structure = ()
)

(define-dialogic-framework auction-admision-df as
  ontology = auction-admision-ontology
  content-language = PROLOG
  illocutionary-particles = (inform failure request )
  external-roles = (buyer )
  internal-roles = (aad )
  social-structure = ()
)
```

```
(define-dialogic-framework info-seeking-df as
  ontology = info-seeking-ontology
  content-language = PROLOG
  illocutionary-particles = (request inform failure )
  external-roles = (buyer )
  internal-roles = (llotja DBM )
  social-structure = ()
)

(define-dialogic-framework auction-results-df as
  ontology = auction-results-ontology
  content-language = PROLOG
  illocutionary-particles = (inform )
  external-roles = (buyer )
  internal-roles = (AB )
  social-structure = ()
)

(define-dialogic-framework rc-programming-df as
  ontology = rc-programming-ontology
  content-language = PROLOG
  illocutionary-particles = (inform request )
  external-roles = (buyer )
  internal-roles = (rc )
  social-structure = ()
)

(define-dialogic-framework auction-room-df as
  ontology = auction-room-ontology
  content-language = PROLOG
  illocutionary-particles = (request inform failure )
  external-roles = ()
  internal-roles = (rc AB )
  social-structure = ()
)

(define-ontology buyer-admission-ontology as
  (open: -> boolean)
  (login:String*String -> boolean)
  (deny:int -> boolean)
  (accept: -> boolean)
  (close: -> boolean)
)

(define-ontology rc-programming-ontology as
```

```
  (open: -> boolean)
  (programbid:String*float*int -> boolean)
  (priceStopped:String*int*int -> boolean)
  (numberBoxes:String*int -> boolean)
  (close: -> boolean)
)

(define-ontology good-registering-ontology as
  (open: -> boolean)
  (newgood:String*String*String*int*float*int*String*String -> boolean)
  (oldgood:String*String*String*int*float*int*String*String -> boolean)
  (close: -> boolean)
)

(define-ontology auction-room-ontology as
  (startauction:int -> boolean)
  (startround:String*int*float*int -> boolean)
  (offer:String*float -> boolean)
  (bid:String*String*int -> boolean)
  (sold:String*String*float*int -> boolean)
  (bidError:String -> boolean)
  (withdrawn:String*float*int -> boolean)
  (mot:String*String -> boolean)
  (collision:String*String list -> boolean)
  (stopprice:String*int*int -> boolean)
  (endauction:int -> boolean)
  (numboxes:int -> boolean)
  (cancelRound: -> boolean)
)

(define-ontology masfit-ontology as
  (datatype infoLlotja = InfoLlotja of string * string)
  (open: -> boolean)
  (close: -> boolean)
  (login:String*String -> boolean)
  (deny:int -> boolean)
  (accept: -> boolean)
  (loginauction:String*String -> boolean)
  (acceptauction: String * String -> boolean)
  (query:String -> boolean)
  (result:String -> boolean)
  (error:int -> boolean)
  (registerLlotja: infoLlotja -> boolean)
  (activeLlotges: -> booelan)
  (llotges: infoLlotja list -> boolean)
```

```
  (newgood:String*String*String*int*float*int*String*String -> boolean)
  (oldgood:String*String*String*int*float*int*String*String -> boolean)
  (programbid:String*float*int -> boolean)
  (priceStopped:String*int*int -> boolean)
  (numberBoxes:String*int -> boolean)
  (startauction:int -> boolean)
  (startround:String*int*float*int -> boolean)
  (offer:String*float -> boolean)
  (bid:String*String*int -> boolean)
  (sold:String*String*float*int -> boolean)
  (bidError:String -> boolean)
  (withdrawn:String*float*int -> boolean)
  (mot:String*String -> boolean)
  (collision:String*String list -> boolean)
  (stopprice:String*int*int -> boolean)
  (endauction:int -> boolean)
  (numboxes:int -> boolean)
  (cancelRound: -> boolean)
)

(define-ontology auction-results-ontology as
  (open: -> boolean)
  (sold:String*String*float*int -> boolean)
  (withdrawn:String*float*int -> boolean)
  (close: -> boolean)
)

(define-ontology info-seeking-ontology as
  (datatype infoLlotja = InfoLlotja of string * string)
  (open: -> boolean)
  (query:String -> boolean)
  (result:String -> boolean)
  (error:int -> boolean)
  (registerLlotja: infoLlotja -> boolean)
  (activeLlotges: -> booelan)
  (llotges: infoLlotja list -> boolean)
  (close: -> boolean)
)

(define-ontology auction-admision-ontology as
  (open: -> boolean)
  (loginauction:String*String -> boolean)
  (deny:int -> boolean)
  (acceptauction: String * String -> boolean)
  (close: -> boolean)
```

```
)
```

A.2 Conferen Centre

```
(define-institution eInsititution as
  dialogic-framework = CC-DF
  performative-structure = CC-PS
  norms = (app-notification)
)

(define-performative-structure CC-PS as
  scenes = (
     (IGS InformationGatheringScene)
     (APS AppointmentCoordinationScene list)
     (ADS AdvertiserScene list)
     (root root_scene)
     (ACS AppointmentCoordinationScene list)
     (CS ContextScene)
     (exit exit_scene)
     (DS DeliveryScene))
  transitions = (
      (T12 OR)
      (T7 AND)
      (T3 OR)
      (T2 OR)
      (T9 OR)
      (T13 AND)
      (T6 OR)
      (T0 AND)
      (T10 AND)
      (T8 OR)
      (T5 OR)
      (T4 OR)
      (T14 AND)
      (T1 OR)
      (T11 OR))
  connections = (
      (IGS T2 (((x IG))))
      (root T8 (((x D))))
      (root T13 (((x AW))))
      (IGS T7 (((x IG)(y IG))))
      (ADS T4 (((x AD))((y IF))))
      (APS T5 (((x pro))))
      (root T3 (((x B))((y IG))))
      (IGS T14 (((y IG))))
```

```
      (ACS T12 (((x pro)))
           ((not (obl ( ?x (request (?x IP) (?z D)
               (push ?personID ?info)) DS)))) )
      (ACS T5 (((y pro))))
      (DS T9 (((x IP))((y D))))
      (IGS T0 (((x IG)(y IG))))
      (ADS T6 (((x AD))((y IF))))
      (APS T11 (((x pro))))
      (CS T1 (((x AW))((y CM))))
      (IGS T4 (((z IG))((v B))))
      (APS T10 (((x pro)(y pro))))
      (CS T2 (((y CM))))
      (T14 CS (((y CM))) 1)
      (T7 IGS (((x IG)(y IG))) 1)
      (T8 DS (((x D))) new)
      (T1 exit (((x AW))((y CM))) 1)
      (T7 ADS (((x AD)(y IF))) new)
      (T5 exit (((x pro))((y pro))) 1)
      (T9 exit (((x IP))((y D))) 1)
      (T3 IGS (((x B))) new)
      (T0 APS (((x pro)(y pro))) new)
      (T12 DS (((x IP))) 1)
      (T14 IGS (((y IG))) 1)
      (T10 ACS (((x pro)(y pro))) new)
      (T6 DS (((x IP))((y IP))) 1)
      (T13 CS (((x AW))) new)
      (T0 IGS (((x IG)(y IG))) 1)
      (T3 IGS (((y IG))) 1)
      (T2 DS (((x IP))((y IP))) 1)
      (T11 DS (((x IP))) 1)
      (T4 exit (((x AD))((y IF))((z IG))((v B))) 1))
  initial-scene = root
  final-scene = exit
)

(define-scene AdvertiserScene as
  roles = (IF AD)
  scene-dialogic-framework = CC-DF
  states = (W3 W2 W4 W1 W0)
  initial-state = W0
  final-states = (W4 W3)
  access-states = ((IF (W0)) (AD (W0)) )
  exit-states = ((IF (W3 W4)) (AD (W3 W4)))
  agents-per-role = (
       (1 <= IF <= 1)
```

```
      (1 <= AD <= 1))
  connections = (
      (W2 W1 (inform (!x AD) (!y IF) (info ?answer))  )
      (W1 W4 (accept (!y IF) (x AD) (activity !event))  )
      (W1 W2 (request (!y IF) (!x AD) (Info ?query))  )
      (W1 W3 (decline (!y IF) (!x AD) (activity !event))  )
      (W0 W1 (propose (?x AD) (?y IF) (activity ?event))  )
  )
)

(define-scene exit_scene as
  roles = (IG D IF pro B CM IP AW AD)
  scene-dialogic-framework = CC-DF
  states = (W0)
  initial-state = W0
  final-states = (W0)
  access-states = ((IG (W0)) (D (W0)) (IF (W0)) (pro (W0)) (B (W0))
       (CM (W0)) (IP (W0)) (AW (W0)) (AD (W0)) )
  exit-states = ((IG (W0)) (D (W0)) (IF (W0)) (pro (W0)) (B (W0))
       (CM (W0)) (IP (W0)) (AW (W0)) (AD (W0)) )
  connections = ( )
)

(define-scene DeliveryScene as
  roles = (D IP)
  scene-dialogic-framework = CC-DF
  states = (W3 W1 W2 W0)
  initial-state = W0
  final-states = (W3)
  access-states = ((D (W0)) (IP (W1 W0)) )
  exit-states = ((D (W3)) (IP (W3 W1)) )
  agents-per-role = (
      (1 <= D <= 1))
  connections = (
      (W2 W1 (inform (!x D) (!y IP) (pushed !info))  )
      (W0 W1 (inform (?x D) (all) (open ))  )
      (W1 W3 (inform (!x D) (all) (close ))  )
      (W2 W1 (failed (!x D) (!y IP) (reason ?r))  )
      (W1 W2 (request (?y IP) (!x D) (push ?personID ?info))  )
  )
)

(define-scene AppointmentCoordinationScene as
  roles = (pro)
  scene-dialogic-framework = CC-DF
```

```
  states = (W0 W4 W6 W1 W2 W3 W5)
  initial-state = W0
  final-states = (W6 W3)
  access-states = ((pro (W0)) )
  exit-states = ((pro (W6 W3)) )
  agents-per-role = (
       (2 <= pro <= 2))
  connections = (
       (W1 W5  )
        (W5 W1 (propose (!x pro) (!y pro)
                  (app-scheudling ?scheudling-x ?timeout-x))  )
       (W2 W1 (propose (!x pro) (!y pro)
                   (app-scheudling ?scheudling-x ?timeout-x))  )
       (W1 W6 (inform (!y pro) (!x pro)
                   (accept-scheudling !scheudling-x))  )
       (W1 W3 (decline (!y pro) (!x pro) (reason ?r))  )
       (W4 W3 (decline (!y pro) (!x pro) (reason ?r))  )
       (W4 W2 (propose (!y pro) (!x pro)
                   (app-scheudling ?scheudling-y ?timeout-y))  )
       (W2 W6 (accept (!x pro) (!y pro)
                   (accept-scheudling !scheudling-y))  )
       (W0 W1 (propose (?x pro) (?y pro)
                   (app-scheudling ?scheudling-x ?timeout-x))  )
       (W2 W3 (decline (!x pro) (!y pro) (reason ?r))  )
       (W1 W2 (propose (!y pro) (!x pro)
                   (app-scheudling ?scheudling-y ?timeout-y))  )
       (W5 W3 (decline (!x pro) (!y pro) (reason ?r))  )
  )
)

(define-scene InformationGatheringScene as
  roles = (IG B)
  scene-dialogic-framework = CC-DF
  states = (W0 W2 W3 W1)
  initial-state = W0
  final-states = (W3)
  access-states = ((IG (W0 W1)) (B (W0)) )
  exit-states = ((IG (W3 W1)) (B (W3)) )
  agents-per-role = (
       (1 <= B <= 1))
  connections = (
       (W2 W1 (inform (!x B) (!y IG) (interested-people ?agents))  )
       (W1 W1 (accept (?y IG) (?z IG) (app-nego ?presentation))  )
       (W1 W1 (inform (?y IG) (!x B) (subscribe ?presentation))  )
       (W1 W1 (propose (?y IG) (?z IG) (app-nego ?presentation))  )
```

```
      (W1 W2 (request (?y IG) (!x B) (interested ?topic))  )
      (W1 W1 (decline (?u IG) (?v IG) (app-nego ?presentation))  )
      (W1 W3 (inform (!x B) (all) (close ))  )
      (W1 W1 (accept (?y IG) (?z IG) (advertisement ?event))  )
      (W1 W1 (propose (?y IG) (?z IG) (advertisement ?event))  )
      (W1 W1 (decline (?u IG) (?v IG) (advertisement ?event))  )
      (W0 W1 (inform (?x B) (all) (open ))  )
  )
)

(define-scene AppointmentProposalScene as
  roles = (pro)
  scene-dialogic-framework = CC-DF
  states = (W0 W3 W5 W1 W6 W4 W2)
  initial-state = W0
  final-states = (W6 W3)
  access-states = ((pro (W0)) )
  exit-states = ((pro (W3 W6)) )
  agents-per-role = (
      (2 <= pro <= 2))
  connections = (
      (W5 W2 (propose (!y pro) (!x pro)
                  (app-topics ?topics-y ?timeout-y))  )
      (W1 W2 (propose (!y pro) (!x pro)
                  (app-topics ?topic-y ?timeout-y))  )
      (W1 W4 !time-out-x )
      (W1 W6 (inform (!y pro) (!x pro) (accept-topics ?topic-x))  )
      (W2 W3 (decline (!x pro) (!y pro) (reason ?r))  )
      (W0 W1 (propose (?x pro) (?y pro)
                   (app-topics ?topic-x  ?timeout-x))  )
      (W2 W5 !time-out-y)
      (W5 W3 (decline (!y pro) (!x pro) (reason ?r))  )
      (W4 W1 (propose (!x pro) (!y pro)
                   (app-topics ?topic-x timeout-x))  )
      (W1 W3 (decline (!y pro) (!x pro) (reason ?r))  )
      (W2 W1 (propose (!x pro) (!y pro)
                   (app-topics ?topic-x timeout-x))  )
      (W2 W6 (inform (!x pro) (!y pro) (accept-topics ?topic-y))  )
      (W4 W3 (decline (!x pro) (!y pro) (reason ?r))  )
  )
)

(define-scene root_scene as
  roles = (B D IG AW)
  scene-dialogic-framework = CC-DF
```

```
  states = (W0)
  initial-state = W0
  final-states = (W0)
  access-states = ((B (W0)) (D (W0)) (IG (W0)) (AW (W0)) )
  exit-states = ((B (W0)) (D (W0)) (IG (W0)) (AW (W0)) )
  connections = ( )
)

(define-scene ContextScene as
  roles = (CM AW)
  scene-dialogic-framework = CC-DF
  states = (W0 W2 W1)
  initial-state = W0
  final-states = (W2)
  access-states = ((CM (W0 W1)) (AW (W0)) )
  exit-states = ((CM (W2 W1)) (AW (W2)) )
  agents-per-role = (
       (1 <= AW <= 1))
  connections = (
       (W1 W2 (inform (!x AW) (all) close())  )
       (W0 W1 (inform (?x AW) (all) (open))  )
       (W1 W1 (inform (!x AW) (?y CM) (context_info ?info))  )
       (W1 W1 (request (?x CM) (!y AW)
                    (contextSubscription ?presentation))  )
  )
)

(define-dialogic-framework CC-DF as
  ontology = CC-ontology
  content-language = PROLOG
  illocutionary-particles = (inform accept request failed decline propose)
  external-roles = (IF AD pro IG IP CM )
  internal-roles = (D AW B )
  social-structure = ()
)

(define-ontology CC_ontology as
  (datatype name = Name of String)
  (datatype topic = Topic of String)
  (datatype location = Location of String)
  (datatype scheudling = Scheudling of String)
  (datatype eventType = EventType of String)
  (datatype event = Event of name * eventType * location * topic list)
  (datatype presentation = Presentation of name * topic list * event list)
  (open: boolean)
```

```
  (close: boolean)
  (context_info: String -> boolean)
  (contextSubscription: presentation -> boolean)
  (push: String * String -> boolean)
  (app-topics: topic list * int -> boolean)
  (accept-topics: topic list -> boolean)
  (app-scheudling: scheudling * int -> boolean)
  (accept-scheudling: scheudling -> boolean)
  (reason: String -> boolean)
  (pushed: String -> boolean)
  (advertisement: event -> boolean)
  (subscribe: presentation ->boolean)
  (advertisement: event -> boolean )
  (app-nego:  presentation -> boolean  )
  (interested-people: AgentID list -> boolean)
  (interested: topic -> boolean)
  (info: String -> boolean)
  (activity: event -> boolean)
)

(define-norm app-notification as
  antecedent =
       (((ACS (accept (?x pro) (?y pro) (app ?date-y)))))
  defeasible-antecedent =
       ((DS (request (?x IP) (?z D) (push ?personID ?info))))
  consequent =
       ( (obl ( ?x (request (?x IP) (?z D) (push ?personID ?info)) DS)))
)
```

Bibliography

[3APL, URL] 3APL (URL). 3apl url. http://www.cs.uu.nl/3apl/.

[Apt, 1997] Apt, K. R. (1997). *From Logic Programming to Prolog*. Prentice-Hall, U.K.

[Arcos and Plaza, 2002] Arcos, J. L. and Plaza, E. (2002). Context-aware personal information agents. *International Journal on Cooperative Information Systems*, 11(3):245–264.

[Ashri and Luck, 2001] Ashri, R. and Luck, M. (2001). Towards a layered approach for agent infrastructure: the right tools for the right job. In *Second International Workshop on Infrastructure for Agents, MAS, and Scalable MAS*.

[Austin, 1962] Austin, J. L. (1962). *How to Do Things With Words*. Oxford University Press.

[Barbuceanu and Fox, 1995] Barbuceanu, M. and Fox, M. S. (1995). Cool: A language for describing coordination in multi-agent systems. In *Proceedings of the First International Conference in Multi-Agent Systems (ICMAS-95)*, pages 17–24. AAAI Press.

[Bauer et al., 2001] Bauer, B., Mller, J. P., and Odell, J. (2001). Agent uml: A formalism for specifying multiagent software systems. *International Journal of Software Engineering and Knowledge Engineering*, 11(3):207–230.

[Bellifemine et al., 2002a] Bellifemine, F., Caire, G., Trucco, T., and Rimassa, G. (2002a). *JADE Administrator's guide.* CSELT, TILab, http://jade.cselt.it.

[Bellifemine et al., 2002b] Bellifemine, F., Caire, G., Trucco, T., and Rimassa, G. (2002b). *JADE Programmer's Guide.* CSELT, TILab, http://jade.cselt.it.

[Bellifemine et al., 2001] Bellifemine, F., Poggi, A., and Rimassa, G. (2001). Developing multi-agent systems with jade. In Castelfranchi, C. and Lesperance, Y., editors, *Intelligent Agents VII*, number 1571 in Lecture Notes in Artificial Intelligence, pages 89–103. Springer-Verlag.

[Bushnell and Oren, 1993] Bushnell, J. and Oren, S. (1993). Two dimensional auctions for efficient franchising of public monopolies. Technical Report ERL-93-41, University of California, Berkeley.

[Carriero and Gelernter, 1989] Carriero, N. and Gelernter, D. (1989). Linda in Context. *Comm. of the ACM*, 32(4):444–458.

[Chauhan, 1997] Chauhan, D. (1997). *JAFMAS: A Java-based Agent Framework for Multiagent Systems Development and Implementation*. PhD thesis, ECECS Department, University of Cincinnati.

[Clearwater, 1995] Clearwater, S. (1995). *Market-Based Control: A Paradigm for Distributed Resource Allocation*. World Scientific Press.

[Corkill and Lesser, 1983] Corkill, D. D. and Lesser, V. (1983). The use of meta-level control for coordination in a distributed problem solving network. In Bond, A. H. and Gasser, L., editors, *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, pages 748–756. Karlsruhe, Federal Republic of Germany, Morgan Kaufmann Publishers.

[Cost et al., 1999] Cost, R. S., Chen, Y., Finin, T., Labrou, Y., and Peng, Y. (1999). Modeling agent conversations with colored petri nets. In *AGENTS'99 Workshop on Specifying and Implementing Conversation Policies*.

[de la Cruz, 2001] de la Cruz, D. (2001). Islander un editor d'institucions electròniques. Master's thesis, Universitat Autonoma de Barcelona.

[Dellarocas and Klein, 1999] Dellarocas, C. and Klein, M. (1999). Civil agent societies: Tools for inventing open agent-mediated electronic marketplaces. In *Proceedings ACM Conference on Electronic Commerce (EC-99)*.

[Dignum, 2002] Dignum, F. (2002). Abstract norms and electronic institutions. In *Proceedings of International Workshop on Regulated Agent-Based Social Systems: Theories and Applications (RASTA'02)*.

[Dignum and Greaves, 2000] Dignum, F. and Greaves, M. (2000). Issues in agent communication: An introduction. In Dignum, F. and Greaves, M., editors, *Issues in Agent Communication*, volume 1916 of *Lecture Notes in Computer Science*, pages 1–16. Springer Verlag.

[Diller, 1990] Diller, A. (1990). *Z An Introduction to Formal Methods*. John Wiley & Sons, Inc.

[d'Inverno et al., 1998] d'Inverno, M., Kinny, D., and Luck, M. (1998). Interaction protocols in agentis. In *Proceedings of the Third International Conference on Multi-agent Systems (ICMAS-98)*, pages 112–119.

[Esteva et al., 2002a] Esteva, M., de la Cruz, D., and Sierra, C. (2002a). Islander: an electronic institutions editor. In *Proceedings of The First International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2002)*, pages 1045–1052.

[Esteva and Padget, 2000] Esteva, M. and Padget, J. (2000). Auctions without auctioneers: distributed auction protocols. In Moukas, A., Sierra, C., and Ygge, F., editors, *Agent-mediated Electronic Commerce II*, volume 1788 of *Lecture Notes in Artificial Intelligence*, pages 20–38. Springer Verlag.

[Esteva et al., 2002b] Esteva, M., Padget, J., and Sierra, C. (2002b). Formalizing a language for institutions and norms. In Tambe, M. and Meyer, J.-J., editors, *Intelligent Agents VIII*, volume 2333 of *Lecture Notes in Artificial Intelligence*, pages 348–366. Springer Verlag.

[Esteva et al., 2001] Esteva, M., Rodríguez-Aguilar, J. A., Sierra, C., Arcos, J. L., and Garcia, P. (2001). *Agent-mediated Electronic Commerce: The European AgentLink Perspective*, chapter On the Formal Specification of Electronic Institutions, pages 126–147. Number 1991 in Lecture Notes in Artificial Intelligence. Springer-Verlag.

[Ferber and Gutknetch, 1998] Ferber, J. and Gutknetch, O. (1998). A meta-model for the analysis of organizations in multi-agent systems. In *Proceedings of the Third International Conference on Multi-Agent Systems (ICMAS-98)*, pages 128–135.

[Finin et al., 1995] Finin, T., Labrou, Y., and Mayfield, J. (1995). Kqml as an agent communication language. In Bradshaw, J., editor, *Software Agents*. MIT Press, Cambridge. invited chapter.

[FIPA, 1997] FIPA (1997). Specification part 2: Agent communication language. Technical report, Foundation for Intelligent Physical Agents.

[FIPA, 2001] FIPA (2001). Fipa agent management specification. Technical Report XC00023H, Foundation for Intelligent Physical Agents, http://www.fipa.org, Geneva, Switzerland.

[FIPA, 2002] FIPA (2002). Ffipa agent message transport service specification. Technical report, Foundation for Intelligent Physical Agents, http://www.fipa.org, Geneva, Switzerland.

[FishMarket, URL] FishMarket (URL). The Fishmarket Project. http://-www.iiia.csic.es/Projects/fishmarket.

[Forgy, 1982] Forgy, C. L. (1982). Rete: A fast algorithm for the many pattern/ many object pattern match problem. *Artificial Intelligence*, 19:17–37.

[Franklin and Reiter, 1996] Franklin, M. and Reiter, M. (1996). The Design and Implementation of a Secure Auction Service. *IEEE Transactions on Software Engineering*, 22(5):302–312.

[Gagliano et al., 1995] Gagliano, R. A., Fraser, M. D., and Schaefer, M. E. (1995). Auction allocation of computing resources. *Communications of the ACM*, 38(6):88–102.

[Galan, 2000] Galan, A. K. (2000). *JiVE: JAFMAS integrated Visual Environment*. PhD thesis, University of Cincinnati.

[Garcia et al., 1998] Garcia, P., Giménez, E., Godo, L., and Rodríguez-Aguilar, J. A. (1998). Possibilistic-based design of bidding strategies in electronic auctions. In *The 13th biennial European Conference on Artificial Intelligence (ECAI-98)*.

[Gasser et al., 1987] Gasser, L., Braganza, C., and Herman, N. (1987). *Distributed Artificial Intelligence*, chapter MACE: A flexible test-bed for distributed AI research, pages 119–152. Pitman Publishers.

[Genesereth and Ketchpel, 1994] Genesereth, M. R. and Ketchpel, S. P. (1994). Software agents. *Communications of the ACM, Special Issue on Intelligent Agents*, 37(7):48–53.

[Giunchiglia et al., 2002] Giunchiglia, F., Mylopoulos, J., and A-Perini (2002). The tropos software development methodology: Processes, models and diagrams. In *AAMAS'02 Workshop on Agent Oriented Software Engineering (AOSE-2002)*, pages 63–74.

[Greaves et al., 2000] Greaves, M., Holmback, H., and Bradshaw, J. (2000). What is a conversation policy? In *Issues in Agent Communication*, number 1916 in Lecture Notes in Artificial Intelligence, pages 118–131. Springer-Verlag.

[Hewitt, 1986] Hewitt, C. (1986). Offices are open systems. *ACM Transactions of Office Automation Systems*, 4(3):271–287.

[Hoare, 1985] Hoare, C. A. R. (1985). *Communcating Sequential Processes*. Prentice Hall.

[Holzmann, 1997] Holzmann, G. J. (1997). The model checker spin. *IEEE Transactions on Software Engineering*, 25(3).

[Howden et al., 2001] Howden, N., Romquist, R., Hodgson, A., and Lucas, A. (2001). Jack intelligent agents - summary of an agent infrastructure. In *Proceedings of the Fifth InternationalConference on Autonomous Agents*.

[Huberman and Clearwater, 1995] Huberman, B. A. and Clearwater, S. (1995). A multi-agent system for controlling builging environments. In *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95)*, pages 171–176. AAAI Press.

[Huguet et al., 2002] Huguet, M.-P., Esteva, M., Phelps, S., Sierra, C., and Wooldridge, M. (2002). Model checking electronic institutions. In *ECAI Workshop on Model Checking and Artificial Intelligence (MoChart-2002)*.

[Iglesias et al., 1999] Iglesias, C. A., Garijo, M., and Gonzalez, J. C. (1999). A survey of agent-oriented methodologies. In Muller, J. P., Singh, M., and Rao, A. S., editors, *Intelligent Agents V*, Lecture Notes in Artificial Intelligence. Springer-Verlag.

[ISLANDER, URL] ISLANDER (URL). ISLANDER editor. http://e-institutor.iiia.csic.es/e-institutor/software/islander.html.

[JADE, URL] JADE (URL). The Java Agent Development Framework. http://jade.cselt.it.

[Jennings, 2000] Jennings, N. R. (2000). On agent-based software engineering. *Artificial Intelligence*, 117(2):277–296.

[Jennings, 2001] Jennings, N. R. (2001). An agent-based approach for building complex software systems. *Comms. of the ACM*, 44(4):35–41.

[Jennings et al., 1998] Jennings, N. R., Sycara, K., and Wooldridge, M. (1998). A roadmap of agent research and development. *Autonomous Agents and Multi-agent Systems*, 1:275–306.

[Jennings and Wooldridge, 1998] Jennings, N. R. and Wooldridge, M. J. (1998). Applications of intelligent agents. In Jennings, N. R. and Wooldridge, M. J., editors, *Agent Technology: Foundations, Applications, and Markets*, pages 3–28. Springer-Verlag: Heidelberg, Germany.

[JESS, URL] JESS (URL). JESS Webpage. http://herzberg.ca.sandia.gov/jess.

[Juan et al., 2002] Juan, T., Pierce, A., and Sterling, L. (2002). Roadmap: Extending the gaia methodology for complex open systems. In *Proceedings of The First International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2002)*, pages 3–10.

[Koning et al., 1998] Koning, J.-L., Franois, G., and Demazeau, Y. (1998). Formalization and pre-validation for interaction protocols in multiagent systems. In Prade, H., editor, *Proceedings of the 13th European Conference on Artificial Intelligence*, pages 298–302. John Wiley & Sons, Ltd.

[Labrou et al., 1999] Labrou, Y., Finin, T., and Peng, Y. (1999). Agent communication languages: The current landscape. *IEEE Intelligent Systems*, 14(2):45–52.

[Lamport et al., 1982] Lamport, L., Shostak, R., and Pease, M. (1982). The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401.

[Lynch, 1996] Lynch, N. (1996). *Distributed Algorithms*. Morgan Kaufmann. ISBN 1-55860-348-4.

[Martín et al., 2000] Martín, F. J., Plaza, E., and Rodríguez-Aguilar, J. A. (2000). An infrastructure for agent-based systems: An interagent approach. *International Journal of Intelligent Systems*, 15(3):217–240.

[MASFIT, URL] MASFIT (URL). The MASFIT project. http://www.masfit.net.

[Mazoui et al., 2002] Mazoui, H., Fallah, A. E., and Haddad, S. (2002). Open protocol design for complex interactions in multi-agent systems. In *Proceedings of The First International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2002)*, pages 517–526.

[Milner, 1980] Milner, R. (1980). *A Calculus of Communicating Systems*. Springer, Berlin, 1 edition.

[Milner, 1991] Milner, R. (1991). The Polyadic π-Calculus: a Tutorial. Preprint of Proceedings International Summer School on Logic and Algebra of Specification.

[Milner, 1999] Milner, R. (1999). *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press.

[Nodine and Unruh, 1999] Nodine, M. H. and Unruh, A. (1999). Constructing robust conversation policies in dynamic agent communities. In *AGENTS'99 Workshop on Specifying and Implementing Conversation Policies*.

[Noriega, 1997] Noriega, P. (1997). *Agent-Mediated Auctions: The Fishmarket Metaphor*. Number 8 in IIIA Monograph Series. Institut d'Investigació en Intel.ligència Artificial (IIIA). PhD Thesis.

[North, 1990] North, D. (1990). *Institutions, Institutional Change and Economics Perfomance*. Cambridge U. P.

[Nwana et al., 1999] Nwana, H. S., Ndumu, D. T., Lee, L. C., and Collis, J. C. (1999). ZEUS: a toolkit and approach for building distributed multi-agent systems. In Etzioni, O., Müller, J. P., and Bradshaw, J. M., editors, *Proceedings of the Third International Conference on Autonomous Agents (Agents'99)*, pages 360–361, Seattle, WA, USA. ACM Press.

[Odell et al., 2000] Odell, J., Parunak, H., and Bauer, B. (2000). Extending uml for agents. In *Agent-Oriented Information Systems Workshop at AAAI 2000*, pages 3–17.

[Ontañón and Plaza, 2002] Ontañón, S. and Plaza, E. (2002). A bartering approach to improve multiagent learning. In *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2002)*, pages 386–393. ACM press.

[Padget, 2001] Padget, J. (2001). Modelling simple market structures in process algebras with locations. In Moreau, L., editor, *AISB'01 Symposium on Software Mobility and Adaptive Behaviour*, pages 1–9. The Society for the Study of Artificial Intelligence and the Simulation of Behaviour.

[Padget and Bradford, 1998] Padget, J. and Bradford, R. (1998). A π-calculus model of a spanish fish market. In Sierra, C. and Noriega, P., editors, *Agent-Mediated Electronic Trading*, number 1571 in Lecture Notes in Artificial Intelligence, pages 166–188. Springer-Verlag.

[Padget and Bradford, 1999] Padget, J. and Bradford, R. (1999). A π-calculus model of the spanish fishmarket. In *Proceedings of AMET'98*, volume 1571 of *Lecture Notes in Artificial Intelligence*, pages 166–188. Springer Verlag.

[Padgham and Winikoff, 2002] Padgham, L. and Winikoff, M. (2002). Prometheus: A methodology for developing intelligent agents. In *AAMAS'02 Workshop on Agent Oriented Software Engineering (AOSE-2002)*, pages 135–145.

[Parunak, 2000] Parunak, H. V. D. (2000). A practitioners? review of industrial agent applications. *Autonomous Agents and Multi-Agent Systems*, 3(4):389–407.

[Parunak and Odell, 2002] Parunak, H. V. D. and Odell, J. (2002). Representing social structures in uml. In Wooldridge, M., Ciancarini, P., and Weiss, G., editors, *Agent-Oriented Software Engineering Workshop II*, volume 2222 of *Lecture Notes in Computer Science*, pages 1–16.

[Pattison et al., 1987] Pattison, H. E., Corkill, D. D., and Lesser, V. R. (1987). *Distributed Artificial Intelligence*, chapter Instantiating Descriptions of Organizational Structures, pages 59–96. Pitman Publishers.

[Pierce, 1996] Pierce, B. C. (1996). Foundational calculi for programming languages. In Tucker, A. B., editor, *Handbook of Computer Science and Engineering*, chapter 139. CRC Press.

[Pierce and Turner, 1997] Pierce, B. C. and Turner, D. N. (1997). Pict: A Programming Language Based on the Pi-Calculus. Technical Report 476, Indiana University.

[Poslad et al., 2000] Poslad, S., Buckle, P., and Hadingham, R. (2000). The fipa-os agent platform: Open source for open standards. In *Proc. of the 5th International Conference and Exhibition on the Practical Application of Intelligent Agents and MultiAgents*, pages 355–368.

[Rodríguez-Aguilar, 2001] Rodríguez-Aguilar, J. A. (2001). *On the Design and Construction of Agent-mediated Electronic Institutions*. PhD thesis, Universitat Autonoma de Barcelona. Also to appear in IIIA monography series.

[Rodríguez-Aguilar et al., 1998] Rodríguez-Aguilar, J. A., Martín, F. J., Noriega, P., Garcia, P., and Sierra, C. (1998). Towards a test-bed for trading agents in electronic auction markets. *AI Communications*, 11(1):5–19.

[Rodríguez-Aguilar et al., 2000] Rodríguez-Aguilar, J. A., Martín, F. J., Noriega, P., Garcia, P., and Sierra, C. (2000). Towards a formal specification of complex social structures in multi-agent systems. In Padget, J. A., editor, *Collaboration between Human and Artificial Societies*, volume 1624 of *Lecture Notes in Artificial Intelligence*, pages 284–300. Springer-Verlag.

[Rodríguez-Aguilar et al., 1997] Rodríguez-Aguilar, J. A., Noriega, P., Sierra, C., and Padget, J. (1997). Fm96.5 a java-based electronic auction house. In *Second International Conference on The Practical Application of Intelligent Agents and Multi-Agent Technology(PAAM'97)*, pages 207–224.

[Searle, 1969] Searle, J. R. (1969). *Speech acts*. Cambridge U.P.

[Sibertin et al., 2000] Sibertin, C., Hamachi, C., and J.Cardoso (2000). Communication protocols as a first-class components of multiagent systems. In *Proceedings of the Fourth International Conference on Multi-agent Systems (ICMAS-00)*, pages 437–438.

[Sibertin-Blanc, 2001] Sibertin-Blanc, C. (2001). Cooperative objects: Principles, use and implementation. *Lecture Notes in Computer Science: Concurrent Object-Oriented Programming and Petri Nets, Advances in Petri Nets*, 2001:216–246.

[SICS, 2000] SICS (2000). SICStus Prolog User's Manual. Swedish Institute of Computer Science, available at `http://www.sics.se/isl/sicstus2.html#Manuals`.

[Sterling and Shapiro, 1994] Sterling, L. and Shapiro, E. (1994). *The Art of Prolog: Advanced Programming Techniques*. MIT Press, 2nd edition.

[Tremblay and Sgrenson, 1985] Tremblay, J.-P. and Sgrenson, P. G. (1985). *The theory and practice of compiler writing*. McGraw-Hill. ISBN 0-07-065161-2,Pag 275-286.

[Varian, 1995] Varian, H. R. (1995). Economic mechanism design for computerized agents. In *First USENIX Workshop on Electronic Commerce*.

[Vasconcelos et al., 2003] Vasconcelos, W. W., Robertson, D., Sierra, C., Esteva, M. Sabater, J., and M., W. (2003). Rapid prototyping of large multi-agent systems through logic programming. *Annals of Mathematics and Artifical Intelligence*. (to appear).

[Vasconcelos et al., 2002a] Vasconcelos, W. W., Sabater, J., Sierra, C., and Querol, J. (2002a). Skeleton-based Agent Development for Electronic Institutions. In *Proc. 1st Int'l Joint Conf. on Autonomous Agents & Multi-Agent Systems (AAMAS 2002)*, Bologna, Italy. ACM, U.S.A.

[Vasconcelos et al., 2002b] Vasconcelos, W. W., Sierra, C., and Esteva, M. (2002b). An Approach to Rapid Prototyping of Large Multi-Agent Systems. In *Proc. 17th IEEE Int'l Conf. on Automated Software Engineering (ASE 2002)*, Edinburgh, UK. IEEE Computer Society, U.S.A.

[Vázquez-Salceda and Dignum, 2003] Vázquez-Salceda, J. and Dignum, F. (2003). Modelling electronic organizations. In *Proceedings of the 3rd International/Central and Eastern European Conference on Multi-Agent Systems*. (to appear).

[Werner, 1987] Werner, E. (1987). *Distributed Artificial Intelligence*, chapter Cooperating Agents: A Unified Theory of Communication and Social Structure, pages 3–36. Pitman Publishers.

[Wooldridge et al., 2002] Wooldridge, M., Fischer, M., Huguet, M.-P., and Parsons, S. (2002). Model checking multiagent systems with mable. In *Proceedings of The First International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2002)*.

[Wooldridge and Jennings, 1999] Wooldridge, M. and Jennings, N. (1999). Software engineering with agents: Pitfalls and pratfalls. *IEEE Internet Computing*, 3(3):20–27.

[Wooldridge et al., 1999] Wooldridge, M., Jennings, N. R., and Kinny, D. (1999). A methodology for agent-oriented analysis and design. In *Proceedings of the Third International Conference on Autonomous Agents (AGENTS'99)*.

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

[Wooldridge and P.Ciancarini, 2001] Wooldridge, M. and P.Ciancarini (2001). Agent-oriented software engineering: The state of the art. In Ciancarini, P. and Wooldridge, M., editors, *Agent-Oriented Software Engineering*, volume 1957 of *Lecture Notes in AI*. Springer Verlag.

[Ygge and Akkermans, 1996] Ygge, F. and Akkermans, H. (1996). Power load management as a computational market. In *Proceedings of the Second International Conference on Multi-Agent Systems (ICMAS-96)*.

[Ygge and Akkermans, 1997] Ygge, F. and Akkermans, H. (1997). Making a case for multi-agent systems. In Boman, M. and de Velde, W. V., editors, *Advances in Case-Based Reasoning*, number 1237 in Lecture Notes in Artificial Intelligence, pages 156–176. Springer-Verlag.