

***METHODOLOGIES
AND SOFTWARE
ENGINEERING FOR
AGENT SYSTEMS***

***The Agent-Oriented
Software Engineering
Handbook***

**Edited by
Federico Bergenti
Marie-Pierre Gleizes
Franco Zambonelli**

Foreword by Katia Sycara

 **Kluwer Academic Publishers**

METHODOLOGIES AND SOFTWARE ENGINEERING FOR AGENT SYSTEMS

*The Agent-Oriented
Software Engineering Handbook*

MULTIAGENT SYSTEMS, ARTIFICIAL SOCIETIES, AND SIMULATED ORGANIZATIONS

International Book Series

Series Editor: Gerhard Weiss, *Technische Universität München*

Editorial Board:

Kathleen M. Carley, Carnegie Mellon University, PA, USA
Yves Demazeau, CNRS Laboratoire LEIBNIZ, France
Ed Durfee, University of Michigan, USA
Les Gasser, University of Illinois at Urbana-Champaign, IL, USA
Nigel Gilbert, University of Surrey, United Kingdom
Michael Huhns, University of South Carolina, SC, USA
Nick Jennings, University of Southampton, UK
Victor Lesser, University of Massachusetts, MA, USA
Katia Sycara, Carnegie Mellon University, PA, USA
Michael Wooldridge, University of Liverpool, United Kingdom

Books in the Series:

CONFLICTING AGENTS: *Conflict Management in Multi-Agent Systems*, edited by Catherine Tessier, Laurent Chaudron and Heinz-Jürgen Müller, ISBN: 0-7923-7210-7

SOCIAL ORDER IN MULTIAGENT SYSTEMS, edited by Rosaria Conte and Chrysanthos Dellarocas, ISBN: 0-7923-7450-9

SOCIALLY INTELLIGENT AGENTS: *Creating Relationships with Computers and Robots*, edited by Kerstin Dautenhahn, Alan H. Bond, Lola Cañamero and Bruce Edmonds, ISBN: 1-4020-7057-8

CONCEPTUAL MODELLING OF MULTI-AGENT SYSTEMS: *The CoMoMAS Engineering Environment*, by Norbert Glaser, ISBN: 1-4020-7061-6

GAME THEORY AND DECISION THEORY IN AGENT-BASED SYSTEMS, edited by Simon Parsons, Piotr Gmytrasiewicz, Michael Wooldridge, ISBN: 1-4020-7115-9

REPUTATION IN ARTIFICIAL SOCIETIES: *Social Beliefs for Social Order*, by Rosaria Conte, Mario Paolucci, ISBN: 1-4020-7186-8

AGENT AUTONOMY, edited by Henry Hexmoor, Cristiano Castelfranchi, Rino Falcone, ISBN: 1-4020-7402-6

AGENT SUPPORTED COOPERATIVE WORK, edited by Yiming Ye, Elizabeth Churchill, ISBN: 1-4020-7404-2

DISTRIBUTED SENSOR NETWORKS, edited by Victor Lesser, Charles L. Ortiz, Jr., Milind Tambe, ISBN: 1-4020-7499-9

AN APPLICATION SCIENCE FOR MULTI-AGENT SYSTEMS, edited by Thomas A. Wagner, ISBN: 1-4020-7867-6

METHODOLOGIES AND SOFTWARE ENGINEERING FOR AGENT SYSTEMS

The Agent-Oriented Software Engineering Handbook

Edited by

Federico Bergenti

Università degli Studi di Parma, Italy

Marie-Pierre Gleizes

*Institut de Recherche en Informatique de Toulouse
(CNRS – INP – UPS), France*

Franco Zambonelli

Università degli Studi di Modena e Reggio Emilia

KLUWER ACADEMIC PUBLISHERS

NEW YORK, BOSTON, DORDRECHT, LONDON, MOSCOW



eBook ISBN: 1-4020-8058-1
Print ISBN: 1-4020-8057-3

©2004 Springer Science + Business Media, Inc.

Print ©2004 Kluwer Academic Publishers
Boston

All rights reserved

No part of this eBook may be reproduced or transmitted in any form or by any means, electronic, mechanical, recording, or otherwise, without written consent from the Publisher

Created in the United States of America

Visit Springer's eBookstore at:
and the Springer Global Website Online at:

<http://www.ebooks.kluweronline.com>
<http://www.springeronline.com>



Contents

Contributing Authors	xi
Foreword	xxi
Introduction	xxvii
Part I Concepts and Abstractions of Agent-Oriented Software Engineering	
Introduction	3
1	
Agent-Based Abstractions for Software Development	5
<i>Mumindar P. Singh</i>	
1. Introduction	5
2. A Brief History of Software Development	7
3. Agents and Multiagent Systems	9
4. Agent-Based Software Development	13
5. Critical Directions	16
6. Conclusions	17
2	
On the Use of Agents as Components of Software Systems	19
<i>Federico Bergenti and Michael N. Huhns</i>	
1. Introduction	19
2. Software Agents vs. Software Components	20
3. Semantically Reusing Agents and Components	25
4. Discussion	30
3	
A Survey on Agent-Oriented Software Engineering Research	33
<i>Jorge J. Gómez-Sanz, Marie-Pierre Gervais and Gerhard Weiss</i>	
1. Introduction	33
2. Analysis	35
3. Design	41
4. Implementation	55
5. Testing	57

6.	More Information	61
7.	Conclusions	61

Part II Methodologies for Agent-Based Systems Development

Introduction	65
--------------	----

4	The Gaia Methodology	69
---	----------------------	----

Luca Cernuzzi, Thomas Juan, Leon Sterling and Franco Zambonelli

1.	Introduction	69
2.	Gaia in a Nutshell	70
3.	Gaia v.2	75
4.	The ROADMAP Methodology	79
5.	Extending Gaia with AUML	84
6.	Open Issues	87
7.	Conclusions	87

5	The Tropos Methodology	89
---	------------------------	----

Paolo Giorgini, Manuel Kolp, John Mylopoulos and Marco Pistore

1.	Introduction	89
2.	Overview	90
3.	Formal Tropos	94
4.	Socially-Based MAS Architectures	98
5.	Goal Models	102
6.	Conclusions	105

6	The MaSE Methodology	107
---	----------------------	-----

Scott A. DeLoach

1.	Introduction	107
2.	Methodology	108
3.	Analysis Phase	108
4.	Design Phase	117
5.	agentTool	122
6.	Applications	124
7.	Comparison with other Methodologies	124

7	A Comparative Evaluation of Agent-Oriented Methodologies	127
---	--	-----

Arnon Sturm and Onn Shehory

1.	Introduction	127
2.	The Evaluation Framework	129
3.	Evaluating Gaia	134
4.	Evaluating Tropos	138
5.	Evaluating MaSE	143
6.	Summary and Conclusion	147

Part III Special-Purpose Methodologies

Introduction	153
8	
The ADELFE Methodology	157
<i>Gauthier Picard and Marie-Pierre Gleizes</i>	
1. Introduction	157
2. ADELFE Methodology Overview	158
3. Preliminary Requirements	161
4. Final Requirements	161
5. Analysis	163
6. Design	165
7. ADELFE Tools	172
8. Comparison with other Methodologies	173
9. Conclusion	174
9	
The MESSAGE Methodology	177
<i>Giovanni Caire, Wim Coulier, Francisco Garijo, Jorge Gómez-Sanz, Juan Pavón, Paul Kearney and Philippe Massonet</i>	
1. Introduction	177
2. The MESSAGE Methodology	178
3. Analysis/Design Travel Agent Case-Study	183
4. Considerations on Low-Level Design	191
5. Evaluation of MESSAGE	193
6. Conclusions	194
10	
The SADDE Methodology	195
<i>Carles Sierra, Jordi Sabater, Jaume Agusti and Pere Garcia</i>	
1. Introduction	195
2. The SADDE Methodology	196
3. A Case Study: The Electricity Market	199
4. Step 1: The EBM	199
5. Step 2: The Electronic Institution	203
6. Step 3: The ABM	206
7. Step 4: Multiagent System	210
8. Cycle P4 through Evolutionary Computing	210
9. Conclusions	214
11	
The Prometheus Methodology	217
<i>Michael Winikoff and Lin Padgham</i>	
1. Introduction	217
2. System Specification	220
3. Architectural Design	222
4. Detailed Design	226
5. Tool Support	228

6.	Experiences with Using Prometheus	230
7.	Related Work	231
8.	Future Work	234
Part IV Tools and Infrastructures for Agent-Oriented Software Engineering		
12		
	The AUML Approach	237
	<i>Marc-Philippe Huget, James Odell and Bernhard Bauer</i>	
1.	Introduction	237
2.	Agent UML Purpose	238
3.	Current Work in Agent UML	239
4.	Future Directions in Agent UML	252
5.	Conclusion	256
13		
	FIPA-Compliant Agent Infrastructures	259
	<i>Fabio Belfemine and Agostino Poggi</i>	
1.	Introduction	259
2.	FIPA	260
3.	FIPA-Compliant Agent Infrastructures	262
4.	JADE	264
5.	Conclusions	272
14		
	Coordination Infrastructures in the Engineering of Multiagent Systems	273
	<i>Andrea Omicini, Sascha Ossowski and Alessandro Ricci</i>	
1.	Introduction	273
2.	Coordination in MAS	274
3.	Infrastructures for MAS Engineering	278
4.	Modelling Coordination Infrastructures with Activity Theory	283
5.	Engineering MAS with Coordination Infrastructures	290
6.	An Example of a Coordination Infrastructure	293
7.	Discussion	295
Part V Non Traditional Approaches to Agent-Oriented Software Engineering		
	Introduction	299
15		
	Engineering Amorphous Computing Systems	303
	<i>Radhika Nagpal and Marco Mamei</i>	
1.	Introduction	303
2.	The Amorphous Computing Model	305
3.	Developmental Biology as an Inspiration	305
4.	Towards Programming Languages	309
5.	Pervasive Computing	315

16

Making Self-Organising Adaptive Multiagent Systems Work 321

Jean-Pierre Georgé, Bruce Edmonds and Pierre Glize

1. Introduction 321
2. Characterization of Emergence in Synthetic Systems 325
3. An Example of a MAS Technology using Emergence 327
4. Flood Forecast by Cooperative Self-Organizing Agents 331
5. Software Engineering Requirements for Self-Organizing MAS 337
6. Conclusion 340

17

Engineering Swarming Systems 341

H. Van Dyke Parunak and Sven A. Brueckner

1. What is Swarming? 341
2. Where would You Want to Use Swarming? 349
3. Why does Swarming Work? 353
4. How can We Apply these Principles in Engineered Systems? 364
5. Conclusion and Prospect 375

18

Online Engineering and Open Computational Systems 377

Martin Fredriksson and Rune Gustavsson

1. Introduction 377
2. Open Computational Systems 379
3. Online Engineering 382
4. Methodological Benchmarking 386
5. Concluding Remarks and Future Work 388

Part VI Emerging Trends and Perspectives

Introduction 393

19

Agents for Ubiquitous Computing 395

Zakaria Maamar, Walter Binder and Boualem Benatallah

1. Introduction 395
2. Examples on Ubiquitous Computing 397
3. Background 398
4. Dimensions of Ubiquitous Computing 401
5. Contributions of Agents to Ubiquitous Computing 404
6. Conclusion 411

20

Agents and the Grid 413

Luc Moreau, Michael Luck, Simon Miles, Juri Papay, Keith Decker and Terry Payne

1. Introduction 413
2. The Grid and Bioinformatics 414
3. Agents in Bioinformatics Grids 416



4.	Agent-Based Service Discovery for Grid Computing	419
5.	Architecture Design	423
6.	Performance Analysis	426
7.	Related Work	428
8.	Conclusion and Future Work	429
21		
	Roadmap of Agent-Oriented Software Engineering	431
	<i>Zahia Guessoum, Massimo Cossentino and Juan Pavón</i>	
1.	Introduction	431
2.	Agents as a New Modeling Paradigm	433
3.	Methods for Building Multiagent Systems	435
4.	Tools for the Implementation, Deployment and Execution	441
5.	Application Opportunities	447
6.	A Roadmap for Agent-Oriented Software Engineering	448
7.	Conclusions	450
	References	451
	Index	503

Contributing Authors

Jaume Agusti

*IIIA – Artificial Intelligence Research Institute
CSIC – Spanish Scientific Research Council
Bellaterra, Catalonia, Spain
agusti@iiia.csic.es*

Bernhard Bauer

*Institute of Computer Science
University of Augsburg
86150 Augsburg, Germany
Bernhard.Bauer@informatik.uni-augsburg.de*

Fabio Bellifemine

*Telecom Italia Lab SpA
Via G. Reiss Romoli 274, 10148 Torino, Italy
Fabio.Bellifemine@tilab.it*

Boualem Benatallah

*The University of New South Wales, Sydney, Australia.
INRIA – Loria, France.
boualem@cse.unsw.edu.au*

Federico Bergenti

*Dipartimento di Ingegneria dell'Informazione
Università degli Studi di Parma
Parco Area delle Scienze 181/A, 43100 Parma, Italy
bergenti@ce.unipr.it*

Walter Binder

*Swiss Federal Institute of Technology, Lausanne, Switzerland.
walter.binder@epfl.ch*



Sven A. Brueckner*Altarum Institute*

sven.brueckner@altarum.org

Giovanni Caire*Telecom Italia Lab SpA**Via G. Reiss Romoli 274, 10148 Torino, Italy*

giovanni.caire@tilab.com

Luca Cernuzzi*Departamento de Ingeniería Electrónica e Informática**Universidad Católica “Nuestra Señora de la Asunción”**Campus Universitario, C.C. 1683, Asunción, Paraguay*

lcernuzz@uca.edu.py

Massimo Cossentino*Istituto di Calcolo e Reti ad Alte Prestazioni**Italian National Research Council*

cossentino@pa.icar.cnr.it

Wim Coulier*Certipost**Centre Monnaie, 1000 Brussels, Belgium*

Wim.COULIER@staff.certipost.be

Keith Decker*Department of Computer and Information Sciences**University of Delaware*

decker@cis.udel.edu

Scott A. DeLoach*Department of Computing & Information Sciences, Kansas State University**234 Nichols Hall, Manhattan, Kansas, USA 66506-2302*

sdeloach@cis.ksu.edu

Bruce Edmonds*CPM – Centre for Policy Modelling, Manchester Metropolitan University**Manchester, M1 3GH, UK*

bruce@cfpm.org

Martin Fredriksson

Societies of Computation

Department of Software Engineering and Computer Science

Blekinge Institute of Technology

Box 520, Ronneby, Sweden.

martin.fredriksson@bth.se

Pere Garcia

IIIA – Artificial Intelligence Research Institute

CSIC – Spanish Scientific Research Council

Bellaterra, Catalonia, Spain

pere@iiia.csic.es

Francisco Garijo

Telefónica I+D

Emilio Vargas, 28043 Madrid, Spain

fgarijo@tid.es

Jean-Pierre Georgé

IRIT – Institut de Recherche en Informatique de Toulouse

118 route de Narbonne, 31062 Toulouse Cedex 4, France

george@irit.fr

Marie-Pierre Gervais

Laboratoire d'Informatique de Paris 6

8 rue du Capitaine Scott, 75015 Paris, France

Marie-Pierre.Gervais@lip6.fr

Paolo Giorgini

Dipartimento di Informatica

Università degli Studi di Trento

Via Sommarive 14, 38050 Povo, Trento, Italy

paolo.giorgini@dit.unitn.it

Marie-Pierre Gleizes

Institut de Recherche en Informatique de Toulouse (CNRS – INP – UPS)

118 route de Narbonne – 31062 Toulouse Cedex France

Marie-Pierre.Gleizes@irit.fr



Pierre Glize

*IRIT – Institut de Recherche en Informatique de Toulouse
118 route de Narbonne, 31062 Toulouse Cedex 4, France
glize@irit.fr*

Jorge J. Gómez-Sanz

*Facultad de Informatica,
Universidad Complutense de Madrid, 28040 Madrid, Spain
jgomez@sip.ucm.es*

Zahia Guessoum

*OASIS (Object and Agents for Simulation and Information Systems) Team
LIP6 (Laboratoire d'Informatique de Paris), Université de Paris 6
Zahia.Guessoum@lip6.fr*

Rune Gustavsson

*Societies of Computation
Department of Software Engineering and Computer Science
Blekinge Institute of Technology
Box 520, Ronneby, Sweden.
rune.gustavsson@bth.se*

Marc-Philippe Huget

*Agent ART Group
Department of Computer Science
University of Liverpool
Liverpool L69 7ZF, United Kingdom
M.P.Huget@csc.liv.ac.uk*

Michael N. Huhns

*Department of Computer Science and Engineering
University of South Carolina
Columbia, SC 29208, USA
huhns@sc.edu*

Thomas Juan

*Department of Computer Science and Software Engineering,
The University of Melbourne, 3010, Victoria, Australia
tj@cs.mu.oz.au*

Paul Kearney

BT Exact

Adastral Park, Martlesham Heath Ipswich IP5 3RE, UK

paul.3.kearney@bt.com

Manuel Kolp

Information Systems Research Unit

School of Management

University of Louvain, Belgium

kolp@isis.ucl.ac.be

Michael Luck

Department of Electronics and Computer Science

University of Southampton

mml@ecs.soton.ac.uk

Zakaria Maamar

Zayed University, Dubai

P.O. Box 19282, Dubai, United Arab Emirates

zakaria.maamar@zu.ac.ae

Marco Mamei

University of Modena and Reggio Emilia

Reggio Emilia, Italy

mamei.marco@unimo.it

Philippe Massonet

CETIC

Rue Clément Ader, 8 B-6041 Gosselies, Belgium

phm@cetic.be

Simon Miles

Department of Electronics and Computer Science

University of Southampton

sm@ecs.soton.ac.uk



Luc Moreau

Department of Electronics and Computer Science

University of Southampton

L.Moreau@ecs.soton.ac.uk

John Mylopoulos

Department of Computer Science

University of Toronto, Canada

jm@cs.toronto.edu

Radhika Nagpal

Massachusetts Institute of Technology

Cambridge, MA, USA 02139

radhi@ai.mit.edu

James Odell

James Odell Associates

3646 W. Huron River Dr.

Ann Harbor, MI 48103, USA

email@jamesodell.com

Andrea Omicini

DEIS, Università di Bologna a Cesena

Via Venezia 52, 47023 Cesena, Italy

mailto:andrea.omicini@unibo.it

Sascha Ossowski

ESCET, Universidad Rey Juan Carlos

Campus de Mostoles, Calle Tulipan s/n, E-28933 Madrid, Spain

mailto:S.Ossowski@escet.urjc.es

Lin Padgham

School of Computer Science and Information Technology

RMIT University

GPO Box 2476V

Melbourne, 3001, Australia

linpa@cs.rmit.edu.au



Juri Papay

*Department of Electronics and Computer Science
University of Southampton
jp@ecs.soton.ac.uk*

H. Van Dyke Parunak

*Altarum Institute
van.parunak@altarum.org*

Juan Pavón

*Universidad Complutense Madrid
Facultad de Informática
28040 Madrid, Spain
jpavon@sip.ucm.es*

Terry Payne

*Department of Electronics and Computer Science
University of Southampton
trp@ecs.soton.ac.uk*

Gauthier Picard

*Institut de Recherche en Informatique de Toulouse (CNRS – INP – UPS)
118 route de Narbonne – 31062 Toulouse Cedex France
picard@irit.fr*

Marco Pistore

*Dipartimento di Informatica
Università degli Studi di Trento
Via Sommarive 14, 38050 Povo, Trento, Italy
marco.pistore@dit.unitn.it*

Agostino Poggi

*Dipartimento di Ingegneria dell'Informazione
Università degli Studi di Parma
Parco Area delle Scienze 181/A, 43100 Parma, Italy
poggi@ce.unipr.it*

Alessandro Ricci

*DEIS, Università di Bologna a Cesena
Via Venezia 52, 47023 Cesena, Italy
mailto:aricci@deis.unibo.it*



Jordi Sabater

*IIIA – Artificial Intelligence Research Institute
CSIC – Spanish Scientific Research Council
Bellaterra, Catalonia, Spain
jsabater@iia.csic.es*

Onn Shehory

*IBM Haifa Research Labs
Haifa, 31905 Israel
onn@il.ibm.com*

Carles Sierra

*IIIA – Artificial Intelligence Research Institute
CSIC – Spanish Scientific Research Council
Bellaterra, Catalonia, Spain
sierra@iia.csic.es*

Munindar P. Singh

*Department of Computer Science
North Carolina State University
Raleigh, NC 27695-7535, USA
singh@ncsu.edu*

Leon Sterling

*Department of Computer Science and Software Engineering,
The University of Melbourne, 3010, Victoria, Australia
leon@cs.mu.oz.au*

Arnon Sturm

*Technion, Israel Institute of Technology
Haifa, 32000 Israel
sturm@tx.technion.ac.il*

Katia Sycara

*School of Computer Science
Carnegie Mellon University
Pittsburgh, PA, 15213, USA
katia@cs.cmu.edu*

Gerhard Weiss

*Institut für Informatik,
TUM Boltzmannstrasse 3, 85748 Garching, Germany
weissg@informatik.tu-muenchen.de*

Michael Winikoff

*School of Computer Science and Information Technology
RMIT University
GPO Box 2476V
Melbourne, 3001, Australia
winikoff@cs.rmit.edu.au*

Franco Zambonelli

*Dipartimento di Scienze e Metodi dell'Ingegneria
Università degli Studi di Modena e Reggio Emilia
Via Allegri 13, 42100 Reggio Emilia, Italy
franco.zambonelli@unimore.it*

Foreword

As information technologies become increasingly distributed and accessible to larger number of people and as commercial and government organizations are challenged to scale their applications and services to larger market shares, while reducing costs, there is demand for software methodologies and applications to provide the following features:

- Richer application end-to-end functionality;
- Reduction of human involvement in the design and deployment of the software;
- Flexibility of software behaviour; and
- Reuse and composition of existing software applications and systems in novel or adaptive ways.

When designing new distributed software systems, the above broad requirements and their translation into implementations are typically addressed by partial complementarities and overlapping technologies and this situation gives rise to significant software engineering challenges. Some of the challenges that may arise are: determining the components that the distributed applications should contain, organizing the application components, and determining the assumptions that one needs to make in order to implement distributed scalable and flexible applications, etc.

Agent-orientation and *Multiagent System* (MAS) principles show much potential at satisfying the above desiderata by means of their inherent modularity and ease with which they can be combined to form new applications. *Agent-Oriented Software Engineering* (AOSE) is distinct from object-orientation in its consideration of agent *goal*, *role*, *context*, and *messages* as first class entities. In addition, agent-orientation encompasses richer semantics, such as speech acts or use of ontologies. Agent-orientation offers higher level abstractions and mechanisms that address additional issues such as knowledge representation and reasoning, coordination, and cooperation among heterogeneous and autonomous parties. These concepts can lead to advanced functionalities,

such as adaptive workflows, matchmaking and brokering services, automated discovery of components. In addition they can provide increased automation, techniques for handling open environments and increased flexibility and fault tolerance. Rich representational capabilities allow more faithful representation of complex organizational processes.

One of the consequences of our characterizations of AOSE is that agents have grounds by which to make inferences about new priorities and consequences as the open, uncertain and dynamic environment changes. That is, the power of inference enables an agent to be adaptable to its changing environment. Another consequence of our agent-orientation is that one of the desiderata of software engineering, namely *predictability* of behaviour changes from two perspectives. From the agent perspective, an agent is now *empowered to infer* its own range of actions, perceptions and expectations for achieving its multiple goals. From the perspective of a MAS observer – which may be another agent as well – the range of actions and behaviours in which the agent may engage can vary greatly based on the unpredictable contexts, the roles of the agents within these contexts, etc. Therefore, predictability estimates should be performed in terms of ample tolerances rather than in terms of precise specification with narrow tolerances. Thus, when designing a MAS an engineer is no longer specifying exactly how an agent will behave, but she is establishing the bounds and tolerances, or an envelope of acceptable behaviour, by which agents may plan their actions and by which observers may judge a MAS behaviours.

The other principal consequence is the necessity of incorporating into the MAS some sort of *planning* behaviour to enable the agents and the overall system to plan in order to achieve their goals, and adapt their behaviour to changing environmental conditions. The endowment of a MAS with planning abilities provides means for (i) composing agent results and behaviour; (ii) monitoring the execution of distributed computations; (iii) fusing the resulting data and control flow; and (iv) automatically determining rollback segments, should distributed computations fail partially.

However, the same qualities of flexibility, autonomy and openness that make AOSE so promising, lead to a number of challenges. MAS operate in an open, unbounded world and thus have imposed upon them the requirements that they must be context-aware, aware of how the environment conditions the interactions that an agent engages in. Openness also refers to the actual implemented agent system itself. Even if an initial MAS design begins with a limited number of agents, new agents or newer versions of agents will most likely be added to the original MAS at a later date, as requirements for the system change and new functionalities are required. Because of this openness there is no single point of resource allocation, synchronization or failure recovery. The environment is dynamic and changing, which challenges any software engineering

paradigm that requires the explicit enumeration of objects and relationships in the environment. The MAS distributed computing environment is uncertain, a characteristic that justifies concerns for partial failure recovery. In distributed computing where there is no centralized point of control, the failure of any one computation, communication link or network node can render the distributed execution state of the MAS application inconsistent, and the resulting inconsistency may be difficult to identify and thus difficult to remedy. As the number of agents in the MAS increases, the dimensionality of the above concerns becomes combinatorial and challenges human perception of control and system predictability.

The above characterization leads to a number of specific issues that MAS must address, unpredictably during their life cycle. The most important of them are:

- **Communication:** considerations include how many different communication interfaces can or should an agent have for effecting its communication with its peers and what are the available underlying network protocols and agent communication languages that will be used.
- **Coordination:** there are a variety of coordination techniques, such as capability based coordination, team-oriented coordination, auction-based coordination, which depend primarily on the task that needs to be performed and the coordination attitude of the agent (e.g., cooperative, self-interested, and deceptive).
- **Environment:** the operating environment can range from the network operating environment, in which considerations focus on how well network protocols are performing (e.g., through put, transport reliability, network connection permanence), to the computational environment in which software capabilities change, to physical and terrain environments of agent-augmented hardware and robots.
- **Functionality:** this is the identification and allocation of agent roles in terms of the services or functions that they contribute.
- **Semantic interoperability:** this expresses the issue that arise when two agents interact. They must be certain when using a vocabulary of terms, that they are using the same concepts with the same relevant inferences of relations as the other communicating agent.

The above characteristics and challenges of agent-oriented software development stretch the limits of current software engineering methodologies, thus creating the need to develop new methodologies and tools. The research community has risen to the challenge by creating in the past years a number of methodologies, such as Gaia, MESSAGE, Prometheus, and Tropos, among

others. The various chapters in this book endeavour to address the challenges of AOSE by providing a variety of methodologies and viewpoints that could guide developers in the various parts of the software lifecycle.

The book is organized in six parts: the first part contains chapters that provide historical and conceptual underpinnings of AOSE. This part has three chapters. The first chapter, “Agent-Based Abstractions for Software Development” by M. Singh, provides a historical perspective and features of the agent metaphor that make it well suited for large scale software development. The second chapter, “On the Use of Agents as Components of Software Systems” by F. Bergenti and M. Huhns, compares agents to software components and highlights the similarities and differences, as well as challenges that software developers face when using agents as system components. The third chapter, “A Survey of Agent-Oriented Software Engineering Research” by J. Gómez-Sanz, M.-P. Gervais, and G. Weiss, provides a survey of existing theories, methodologies and software that can be applied at each stage of agent-oriented development.

Part II comprises chapters about methodologies for agent-oriented software development. This part has four chapters. The first chapter, “The Gaia Methodology” by L. Cernuzzi, T. Juan, L. Sterling and F. Zambonelli describes Gaia, its limitations and proposed extensions to overcome these limitations. The second chapter, “The Tropos Methodology” by P. Giorgini, M. Kolp, J. Mylopoulos and M. Pistore gives an overview of Tropos and in particular the early phases of requirements analysis. The third chapter, “The MaSE Methodology” by S. DeLoach, presents MaSE which is built upon UML to describe models of MAS. The last chapter in this part is “A Comparative Evaluation of Agent-Oriented Methodologies” by A. Sturm and O. Shehory, where a comparison is made of Gaia, Tropos and MaSE. The comparison is performed along the dimensions of concepts and properties, notations and modelling techniques, development process and pragmatics of these methodologies.

Part III consists of chapters that describe special purpose methodologies. The first chapter is “The ADELFE Methodology” by G. Picard and M.-P. Gleizes. This chapter presents ADELFE, a methodology for adaptive MAS where the system’s purpose, and environment are studied during the requirements phase. During analysis, adaptive multiagent technology is used. The second chapter is “The MESSAGE Methodology” by G. Caire, W. Coulier, F. Garijo, J. Gómez-Sanz, J. Pavón, P. Kearney and P. Massonet. The chapter presents MESSAGE, which is a methodology that covers analysis and design phases and uses and extends UML by including concepts of organization, goal, role and task. The third chapter is “The SADDE Methodology” by C. Sierra, J. Sabater, J. Augusti, and P. Garcia. The chapter presents work to help a programmer transition from a set of desired system properties expressed in equations to an actual MAS through tuning parameters in an agent population.

The final chapter of part III is “The Prometheus Methodology” by M. Winikoff and L. Padgham. This chapter describes Prometheus, a methodology that has three phases: (i) specification, where the system’s interface is determined; (ii) determination of systems goals; and use cases, and (iii) detailed design phase where the internal agent details are determined.

Part IV consists of three chapters. The first, “The AUML Approach” by M. Huet, J. Odell and B. Bauer presents Agent UML, a UML-based approach to the development of MAS. The second chapter is “FIPA-Compliant Agent Infrastructure” by F. Bellifemine and A. Poggi which presents JADE, a FIPA-compliant agent infrastructure. The last chapter of this part is “Coordination Infrastructures in the Engineering of Multiagent Systems” by A. Omicini, S. Ossowsky and A. Ricci. The chapter adopts activity theory as a basis for multiagent coordination.

Part V has four chapters. The first, “Engineering Amorphous Computing Systems” by R. Nagpal and M. Mamei studies the issue of engineering robust collective behaviour from large numbers of unreliable components. The second, “Making Self-Organizing Adaptive Multiagent Systems Work” by J.-P. George, B. Edmonds and P. Glize studies how to engineer MAS with desirable emergent properties and sketches an approach to developing such systems by focusing on engineering each agent’s responses to non-cooperative situations it may encounter. The third, “Engineering Swarming Systems” by V. Parunak and S. Bruekner, studies the issues of swarming in terms of self-organization and emergence. It goes on to provide some initial principles of engineering such artificial swarming systems. The last chapter of this part, “Online Engineering and Open Computational Systems” by M. Friedriksson and R. Gustavsson, studies the issue of ambient intelligence and introduces an approach for online engineering.

The final part of the book, Part VI looks towards the future. It includes three chapters. The first, “Agents for Ubiquitous Computing” by Z. Maamar, W. Binder, and B. Benatallah, explores the subject of value-added of software agents to ubiquitous computing environments and discusses the need for new agent engineering approaches. The second chapter, “Agents and the Grid” by L. Moreau, M. Luck, S. Miles, J. Papay, K. Decker, T. Payne discusses the potential value-added of software agents to the Grid and in particular Grid service discovery. The third chapter, “A Roadmap of Agent-Oriented Software Engineering” by Z. Guessoum, M. Cossentino, J. Pavón presents the viewpoint of AgentLink researchers regarding future developments of agent-oriented software methodologies and systems.

Katia Sycara

Introduction

Agents and Multiagent Systems (MAS) have emerged as a powerful technology to face the complexity of a variety of today's IT scenarios. Several industrial experiences already testify to the advantages of using agents in manufacturing processes (Bussmann, 1998; Shen and Norrie, 1999), Web services and Web-based computational markets (Kephart, 2002), and distributed network management (Bieszczad et al., 1998), just to mention a few examples. Further, several research studies advise on the possibility of effectively exploiting agents and MAS as enabling technologies for a variety of future scenarios, i.e., Pervasive Computing (Abelson et al., 2000; Tennenhouse, 2000), Grid computing (Foster and Kesselman, 1999), Semantic Web (Berners-Lee et al., 2001).

However, there is an emergent general understanding that MAS, more than an effective technology, represent indeed a novel general-purpose paradigm for software development (Jennings, 2001; Zambonelli and Parunak, 2003). Agent-based computing promotes designing and developing applications in terms of autonomous software entities (agents), situated in an environment, and which can flexibly achieve their objectives by interacting with one another in terms of high-level protocols and languages. These features are definitely well suited to tackle the intrinsic complexity of developing software in modern scenarios. In fact: *(i)* the autonomy of the application components reflects the intrinsically decentralized nature of modern distributed systems (Tennenhouse, 2000) and can be considered as the natural extension to the notions of modularity and encapsulation for systems that are owned by different stakeholders (Parunak, 1997); *(ii)* the flexible way in which agents operate and interact (both with each other and with the environment) is suited to the dynamic and unpredictable situations in which software is expected to operate today (Zambonelli et al., 2001a); and *(iii)* the concept of agency provides for an unified view of the artificial intelligence results and achievements, which eventually can be used to solve real world problems, by making agents and

MAS act as sound and manageable repositories of intelligent behaviors (Russell and Norvig, 1995).

In the last few years, together with the increasing acceptance of agent-based computing as a novel software engineering paradigm, there have been a great deal of research related to the identification and definition of suitable models, tools, and techniques to support the development of complex software systems in terms of MAS (see chapter 3). These researches, which can be roughly grouped under the term *Agent-Oriented Software Engineering* (AOSE) (Jennings, 2000; Wooldridge, 1997), are endlessly proposing a variety of new metaphors, new formal modeling approaches and techniques, and new development methodologies and tools, specifically suited to the novel agent-oriented paradigm.

This book is an attempt to put together in an integrated and organized way a variety of research results and proposals that, although very diverse, share the same general goal of facilitating the development of complex software systems in terms of MAS. An increasing number of scientific papers on the topic of AOSE can be found in the literature, spread over different conference proceedings, journals, newsletters. Therefore, both newcomers and expert researchers in the areas sometimes have difficulties in navigating all such material. It is our hope that this book will help both researchers and students to get a clue of what is going on in the area of AOSE without having to explore thousand of papers in the existing digital libraries and without getting lost in them.

Of course, we are aware that the research in the area of AOSE is still at its early stages. A few research results are already well-assessed, and several research challenges still need to be faced before AOSE can keep its promises and become not only a widely accepted but also a practically usable paradigm for the development of complex software systems in terms of MAS. For these reasons, in this book, we have clearly separated those concepts and techniques that are already quite assessed and already provide useful results for practical use (Parts I, II, III and IV of this book), from those researches that are instead of a still more investigative nature, and that will try to draw the guidelines for agent systems in the near-/medium- term (parts V and VI of this book). In any case, even for those research results that are of a more assessed nature, we have tried to keep a very critical endeavor, by avoiding to support a specific technology or approach and instead, when necessary, by integrating the presentations of different technologies with neutral comparative evaluations of the techniques.

In particular, the book is organized as follows:

- Part I (Concepts and Abstractions of Agent-Oriented Software Engineering) is introductory, and aims both at clarifying the reasons why agent-based is a suitable approach to the development of complex software

systems (better than existing traditional approaches) and at surveying in a broad way the different facets of AOSE researchs.

- Part II (Methodologies for Agent-Based Software Development) shows three different methodologies (namely, Gaia, Tropos, and MaSE) that have been proposed in the past few years as general-purpose approaches to guide the development of complex MAS, all of which have had great impact in the research community. For the sake of balance, a comparative evaluation of the three methodologies complete this part.
- Part III (Special-Purpose Methodologies) shows four additional methodologies (namely, ADELFE, MESSAGE, SADDE, Prometheus) that, although having had a less general impact as of now, exhibit very interesting characteristics making them very suitable for the development of specific classes of MAS (e.g., adaptive MAS and systems based on intelligent intentional agents) and or specific application areas (e.g., telecommunication applications and agent marketplaces).
- In Part IV (Tools and Infrastructures for Agent-Oriented Software Engineering), we shift the focus from methodologies to infrastructures and tools. In fact, while methodologies drive the process of building a MAS, only the availability of appropriate tools and software infrastructures can make the outcome of this process be a well-engineered software systems. Conceptual tools like the FIPA standard and AUML, and software infrastructures such as tuple-based ones and JADE, presented in this part of the book, are as of now the most promising ones available to engineering and developers.
- Part V (Non Traditional Approaches to Agent-Oriented Software Engineering) put the focus on those innovative approaches to the engineering of agent-based systems that starts from radically different assumptions and concepts than those traditionally adopted in software development, i.e., by relying on self-organization principles and on biologically or physically inspired solutions. The chapters in this part, dealing with swarm intelligence, amorphous computing, adaptive MAS, and online engineering of open systems, may give a rather broad perspective on these novel approaches.
- Part VI (Emerging Trends and Perspectives) is of a more application-oriented nature and focuses on two future scenarios of use of MAS technologies, i.e., the Grid and Ubiquitous Computing. The rationale behind this chapter is to show that, in these emerging scenarios, the abstractions of agent-based systems will be likely to be widely exploited and, accordingly, AOSE techniques will be compulsory needed to promote

the development of reliable and effective applications. On this base, and building on the book as a whole, the final chapter of the book delineates a research roadmap in the area of AOSE.

To conclude this short introduction, we want to emphasize that this book represents a collective effort of the AOSE community, which would not have been possible to realize without the contributions of a number of persons, to which we are greatly indebted. First of all, we thank all the authors who accepted to contribute to this book and to put notable efforts in contributing original chapters summarizing in a readable way their key research results. We thank Katia Sycara, for the foreword to this book. We thank the AgentLink Network of Excellence, and all the persons that in the past few years have participated to the meeting of the Methodologies and Software Engineering Special Interest Group of AgentLink. Without them, we would have never reached that broad vision of the researches in the area that has enabled us to conceive this book. In particular, we would like to thank Michael Luck, coordinator of the AgentLink Network, for having made a great work in collating the strengthening the European research community around the key research themes, such as AOSE. Last but not least, we would really like to thank Kluwer Academic Press, in the persons of Melissa Fearon and Gerhard Weiss (editor of the book series on Multiagent Systems of Kluwer) for having supported this work.

Hope you enjoy reading.

Federico, Marie-Pierre, Franco

I

CONCEPTS AND ABSTRACTIONS OF AGENT-ORIENTED SOFTWARE ENGINEERING

Introduction

The notion of agency was the ultimate end of a long research debate that resulted in the general conclusion that no agreed definition for agent can be found. Still, we can list a number of features that we can use to characterize agency. Many authors enumerated such characteristics and a common subset of such lists, that underpin concepts like autonomy and situatedness, are now at the basis of today agent research. This is sufficient from the scientific point of view, because it provides a framework for promoting a well-funded research on agency. Anyway, it is clear today that it is not sufficient to promote the use of agent technologies in real situations. AOSE took over the big challenge of spreading agent technologies down to everyday software developers and to assist them in exploiting all advantages that agents provide.

The three chapters in this part face this challenge frontally and address directly all issues related to the use of agent technologies in the realization of industrial-strength software systems. In particular:

- Chapter 1, “Agent-Based Abstractions for Software Development” by Munindar P. Singh presents the historical and conceptual basis that readers need to fully appreciate tools and techniques for AOSE;
- Chapter 2, “On the Use of Agents as Components of Software Systems” by Federico Bergenti and Michael N. Huhns discusses major benefits and drawbacks that developers face when choosing agents for the realization of a new software system, instead of more mature technology, e.g., software components; and
- Chapter 3, “A Survey on Agent-Oriented Software Engineering Research” by Jorge J. Gomez-Sanz, Marie-Pierre Gervais and Gerhard Weiss guides readers through existing theories, methods, and software tools that can be applied in each stage of an agent-based development process.

Chapter 1

AGENT-BASED ABSTRACTIONS FOR SOFTWARE DEVELOPMENT

Accommodating Complexity in Open Systems

Munindar P. Singh

Abstract This chapter provides the historical and conceptual basis needed to fully appreciate the tools and techniques for agent-based software development. It begins with a review of the historical development of software technology from stand-alone to open systems. It discusses the major challenge of increasing complexity as software systems become larger and more open. The chapter presents the features of the agent metaphor that make it ideally suited for large-scale open systems development as well as the key technical abstractions that emerge from the agents metaphor and their ramifications on software practice.

1. Introduction

This chapter seeks to provide a conceptual framework and historical perspective to better appreciate the tools and techniques for the engineering of agent-based software. It is by no means a comprehensive survey, which would indeed be impossible within the available space and also unnecessary because others (notably, the chapters in this volume) present more detailed descriptions of specific technical approaches.

It helps to first understand the nature of software development in general. We will confine our attention to business applications, especially those that operate within and across enterprises. Software development for enterprise systems has been notoriously difficult. Computing architectures have gone from centralized to rigidly distributed (as in client-server systems) to fully open. Open architectures are characterized by the fact that they enable autonomous, heterogeneous components to be added and removed dynamically – more on this below. Open architectures are becoming increasingly common with the expansion of e-business. A prominent such architecture involves the use of Web services.

Web Services. The Web services architecture views each component as a *service*, which offers a set of capabilities defined as methods (Curbera et al., 2002a). Services can be invoked and can exchange documents. In Web services, the invocation parameters and results and the exchanged documents in general are structured as specified in the eXtensible Markup Language (XML) (Box et al., 2000). The Web services approach defines public service interfaces wherein the methods and the types of their parameters and results are specified (see <http://www.w3.org/TR/wsdl>). A challenge that is unique to open architectures is the discovery and location of the needed components. In Web services, this is handled through the concept of a *registry* as specified in the Universal Description Discovery and Integration (UDDI) standard (Shaikhali et al., 2003). Service implementations are published (by their implementors and promoters) to a registry. A prospective consumer, that is, anyone seeking a service supporting a particular interface, can contact the registry and attempt to find matching service implementations. The registry will typically convey a list of all service implementations matching the given request. The consumer can then select one of the implementations and invoke it as appropriate.

Whereas the Web services architecture addresses the challenge of discovering service implementations, it has some shortcomings. First, there is no encoding of the semantics of the services and the information they process. Thus matching proceeds on syntactic terms. Second, there is no notion of trustworthiness or fitness for use in a particular environment. The consumers are left to deal with these matters on their own.

However, the Web services approach is fast becoming the de facto standard for structuring large systems. The matters of semantics and trust are handled off-line but even then, there is significant payoff in being able to configure large systems flexibly and dynamically, which is indeed enabled by Web services. Therefore, it is helpful to think of modern agent system development as naturally layering on top of Web services and exploiting their existing features, while augmenting the architecture as necessary. Moreover, because the Web services architecture echoes some key properties of agents, this layering is in many ways quite natural provided we are astute enough not to want to replicate every agent idea within Web services: it would make the resulting systems redundant and conceptually harder to understand, especially for the vast body of software practitioners who are our ultimate audience.

Scientific Applications. Although this chapter limits its attention to business applications, it is important to briefly discuss the emerging convergence of business and scientific applications. Business settings now frequently involve computation-intensive analytical and decision-support components, often with associated dedicated hardware, which are the hallmarks of scientific applica-

tions. Scientific settings now frequently involve considerations such as access control and information modeling, which originated in business settings. The specific technologies used in these arenas are also converging. Previously, business settings involved traditional and federated databases with ontologies and multidatabase query languages, which have led into business processes, as generally understood. And, previously, scientific applications involved a greater emphasis on hand-construction of control and data flows, which have led to what are sometimes termed scientific workflows (Vouk and Singh, 1997). Just as the business side was evolving Web services architectures, nominally running over standard protocols such as HTTP, the scientific side was evolving generic metacomputing frameworks culminating in the so-called Grid architectures (Foster and Kesselman, 1999) (see chapter 20). More recently, the commonalities of the Web services and the Grid architectures have been recognized. To exploit the larger body of work on Web services, the Grid community is developing the Open Grid Services Architecture (Foster et al., 2002), but the intellectual payoffs are likely to benefit both sides.

Along the same lines, the recent IBM initiative on autonomic computing seeks to bring easy configurability and manageability to large-scale computational resources. Autonomic computing ends up incorporating ideas from the Grid and from Web services; indeed, some of the major remaining problems that arise in the context of autonomic computing are those for which agents can be a natural approach (Bigus et al., 2002).

Organization. The rest of this chapter is organized as follows. Section 2 offers a historical perspective on software development with respect to how the computing and communications infrastructure has evolved and how software development has sought to keep pace with it. Section 3 introduces the agents concept. Section 4 discusses some key approaches for the development of agent-based software. Section 5 offers a discussion of some critical emerging directions for research into methodologies and software engineering for agents. Section 6 summarizes our main points.

2. A Brief History of Software Development

The first generation of research in computing was based on centralized architectures and considered the challenges of programming in the small. Although all these challenges may have not been addressed, much progress has been made in techniques for programming data structures and algorithms in a robust and reusable manner. The approaches developed there are still of value because all architectures involve components, which must be robustly programmed.

However, the arrival of distribution in enterprise systems introduced a number of complications. In principle, technologies that support distributed sys-

tems enable information that existed in isolation within enterprises to be combined and put to good use. However, when the low-level challenges of networking were overcome, the move to distributed systems only pointed out the painful reality: that the information resources so connected existed in islands of automation, with little more than a stove-pipe approach to combine them and extract any additional value from them. The killer problem here was heterogeneity, that is, the mismatch of the semantics of the information stored in the various resources. Most often, the semantics was not modeled at all, making it difficult if not impossible to reconcile the various resources with bounded practical effort.

The above problem had not quite been solved when the underlying architecture evolved further. Increasingly, with the expansion of the Internet into the commercial sector, it became clear that software systems should interoperate across enterprise boundaries. The Web is the major example of a substrate on which such open information environments can be built. Whereas the environments vary in their scopes, they are unified by the fact that the components of which they are composed are not only heterogeneous but also autonomous. Components come and go all the time. Thus the exact composition of such an environment is never fixed and never to be taken for granted.

The best approaches for the preceding kind of heterogeneous settings involved the use of static information models or database schemas for the different resources followed by the integration of such models. Schema integration is just barely effective when dealing with heterogeneous information systems, because any changes to the individual schemas renders the integrated schema invalid. When we attempt to apply such an approach to open environments, it simply cannot and does not make sense. The schemas of the components can change rapidly; moreover the constituent parties are under no compulsion to collaborate.

The systems in question are open, because they reflect the openness of the organizations in which arise. In a certain sense, the underlying businesses and the organizations within which computation occurs have not changed due to modern technologies. For example, there clearly were supply chains centuries before there was computing. Even from the computational standpoint, supply chains were somehow accommodated in the systems that emerged during the early days of computing. In a practical sense, however, modern technologies have affected businesses and organizations greatly. One way to understand this dichotomy is as follows. The supply chains that existed in days gone by were fraught with many problems. Early uses of computing helped improve the efficiency of how supply chains were built, executed, and monitored. But they left many shortcomings; specifically, these were rigid approaches and did not handle opportunities and exceptions well. This has greatly limited their effectiveness in practical settings. Further, the improvements in the computing

and communications infrastructure has raised expectations in terms of what kinds of efficiency, flexibility, agility, and robustness are expected.

In other words, while the real world has always been open in the sense described in the above introduction, the technologies that we use to build and manage processes have imposed their own limitations. What is needed now is an approach that avoids these limitations. Enter agents.

3. Agents and Multiagent Systems

Faced with such practical challenges from the environments of today, traditional computer science has little to offer in the way of powerful abstractions and flexible techniques. The problems call for taking a broader view of computation, one which remains scientifically principled but accommodates considerations of the organizations within which computations occur and an understanding of the user needs that these computations seek to serve.

A number of definitions of agents have been propounded over the years. Some definitions emphasized agents in user interfaces, typically with some sort of a persona, typically represented in some suitable graphical manner. Such kinds of agents are not of direct relevance to the present topic. The more promising class of definitions stated that an agent is capable of interacting with other parties. Some definitions were tied to specific implementation technologies such as being based on theorem provers, or using internal data structures corresponding to the so-called mentalist concepts, such as beliefs or knowledge, goals or desires, intentions, and so on. By considering the internal construction of an agent, these definitions restrict their universe of discourse unnecessarily and eliminate the possibility of being applied in open settings, whose participants are heterogeneous. Still other definitions presuppose some notion of the rationality of agents. Such definitions too are inappropriately restrictive, because they presume to incorporate a kind of rationality and conceivably can limit the autonomy of agents (see the discussion of autonomy below).

For the above reasons, a good working definition of an agent is that it is a persistent computational entity that can perceive, reason, act, and communicate (Huhns and Singh, 1998). This definition leaves unspecified the matter of how an agent is constructed, whether it has any beliefs and intentions, and whether it is rational. In this manner, it can apply to a variety of practical computations that are found in open settings. The agents so defined can thus reflect (in an information environment) the autonomy and heterogeneity (in the real world) of the participants whom they represent.

The challenges of modern applications in open systems and the potential flexibility of agents suggests that agents will form an excellent basis for solutions that realize the given applications. But how would we go about building

such solutions? Computer science is a science of abstractions; Devlin makes a similar point recently in the context of what computer scientists ought to be taught (Devlin, 2003). Clearly, to build our solutions, we would need to consider and somehow involve the key abstractions that characterize agents. This brings us back to the discussion of possible definitions for agents.

The recent history of agent technology provides us with a useful hint for a reasonable, pragmatic direction. A few years ago it was quite the in thing in the agents community to offer and debate possible definitions of agents. However, this work fell out of vogue when the research community realized that an agreement on definitions was impossible and likely not even needed. What matters most, as usual, is the purpose for which one is creating a definition. If the purpose is to show how to construct individual agents, it might be reasonable to specify the internal designs of agents. However, if the purpose is to specify an open system generically so that different agents could participate in it, then the emphasis should be on the agents interact rather than how they are constructed. A more radical stance, but along the above lines, is to not worry about formal definitions, but about a test for agenthood (Huhns and Singh, 1999). The test proposed by Huhns and Singh essentially considers how an agent interacts with others or changes its behavior in the presence of others.

When we begin to emphasize interactions, our interest naturally shifts from agents (in other words, single-agent systems) to MAS. Indeed, it is a reasonable position to claim that in practice there are no “interesting” single-agent systems. If you must have agents, you can only have them as part of a MAS. The justification for this point is that if there is no interaction and no openness, you would be equally well-served by the abstractions of traditional computer science. Why introduce agents if you are not going to benefit from their special qualities?

Of course, the whole point of this exercise is to show how agents match up with the requirements of open systems. Recall that we mentioned above that open systems are systems consisting of components that are autonomous, heterogeneous, and dynamic. Now let us consider the following essential abstractions of the concept of agency or alternatively of the agent metaphor.

Autonomy. Autonomy corresponds to the independence of a party to act as it pleases. In its broadest construal, autonomy reflects the political autonomy of the participants in a business transaction. No one can make you sell or buy anything; no one can make you negotiate with or even listen to another party; and, no one can make you use any particular reasoning doctrine. In particular, an autonomous party need not even be “rational” in any external sense, because requiring so would constrain its autonomy.

Assuming the autonomy of the interacting parties has the nice effect that one’s approach, if it accommodates autonomy, can then easily apply to virtu-

ally any situation. Such models can accommodate even some cases that are not a reflection of autonomy in the true sense. For example, infrastructural difficulties might cause a participant to stop responding to messages. Others who can handle autonomous participants can naturally accommodate the vagaries of the infrastructure in the same framework. However, it can be important, for example, to decide how much to trust someone to know whether their behavior as manifested was an exercise of their autonomy or caused by failures in infrastructure.

In general, unfettered computational autonomy would have the same undesirable effects as unfettered autonomy in the real world. Moreover, to make any kind of a complete computational model and to make any kind of reliable predictions, we have to assume that the participants' autonomy is constrained in some way. Typically, constraints on autonomy that are specified based on interactions are termed *protocols*.

Heterogeneity. Heterogeneity corresponds to the independence of the designer of a component to construct the component in any manner. Heterogeneity can apply to the information models of the components or to their process models. Usually, in functioning systems, heterogeneity comes about because of historical reasons. No one sets out to design a heterogeneous system, but large systems end up being heterogeneous. Conversely, when we are laying out the parameters of an open system, it is important not to assume homogeneity of internal constructions. Ultimately, for components that successfully function together, there is some limit to their heterogeneity with respect to each other. That is, there must be a specification of some commonality. In the case of information models, the commonality is captured through a shared ontology (Gruber, 1991; van Harmelen et al., 2002); this is relevant to agents, but is not specific to agents because it also arises wherever heterogeneous information sources are to be combined. In the case of process models, the commonality is captured through an approach such as specifying the exposed *significant events* (Singh, 2003). The idea is that the signatures of the actions of the agents that are of consequence to others are exposed without revealing the internal details of construction. These signatures could potentially be standardized and indeed are, for example, in approaches for the two-phase commitment protocols of distributed database transactions (Gray and Reuter, 1993). Anyone implementing an agent to the specified standard would be required to expose the appropriate significant events but would not reveal proprietary details of internal construction.

Dynamism. Dynamism corresponds to the independence of the administrator of a system to configure it flexibly and to change its configuration as needed without explicitly notifying the relevant parties (the other members of the sys-

tem). Open systems are maximally dynamic, because in principle they need no administrator at all. In practice they do need some administrative functionality for purposes such as monitoring and security, and to restrict participation of potential members based on applicable policies. In principle, these functionalities could be distributed among the members, so there would be no administrator, but usually there are sociopolitical reasons for maintaining a responsible party in the system.

Communications. Any components that share their environment will interact – through the environment as it were. One will make changes to the environment that another will observe. When we conceptualize the components as agents, their interactions take on a distinctive flavor. Clearly, agents can continue to interact through their environment, for example, by bumping into each other if they are robots or by modifying files or databases if they are information agents. However, interactions through the environment have the effect that they can violate the autonomy of the interacting parties. That is, none of the interacting parties may have a choice but to interact. For example, the bumping robots may have no choice but to bump; the bumping can cause one or both of the parties to be knocked off course. Likewise, an information agent may need to modify some data item to complete a task and another information agent may need to read that data item. Observable modifications on data force the other party to note the modified values.

We define *communications* as those interactions that preserve the autonomy of the parties concerned. There is no requirement that communications are planned or that there is a structure of beliefs and intentions to back them up. In this sense, this concept is based on the concept of autonomy; if a kind of interaction is autonomy-preserving for the parties concerned, then it qualifies as a communication. Clearly, both bumps and data modifications can be used as a means of conveying information in a given environment. At the lowest levels, communications will be implemented via some physical (that is, environmental) means, for example, by speaking or by sending packets down a data link. But if it is up to the parties to communicate as and when they please (that is, to send and to receive), then the interaction counts as a communication.

Communication is one of the most important agent abstractions. It is often taken for granted and sometimes confused with low-level data transmission, which is merely the mechanics of communication. The semantics of communication has been intensively studied. Suffice it to say that there are two major kinds of approaches based on mentalist and social concepts, respectively. As explained above, mental concepts deal with the internal construction of agents and so do not apply in open settings. Social concepts have broader applicability. A critical review of the literature is available in (Singh, 1998).

Protocols. Communications are not easily studied in a stand-alone manner. When we do not have access to the internal construction of the communicating agents, it is simpler and more appropriate to characterize combinations of communicative actions. These combinations are protocols, corresponding to the constraints on agent behavior that we alluded to above. In simple terms, a protocol states when and how an agent may communicate with other agents. To give a simplistic business example, a protocol may call for an agent to give a price quote when requested, to pay for items it ordered when they are received, and so on.

Protocols are studied in other branches of computer science, especially networking. Networking protocols, for example, constrain the messages and responses that communicating parties can send to one another. These protocols typically specify the data encodings for various terms, such as headers, to be communicated. Rigid specifications have been thought essential to ensure reliable implementations. However, for agents we wish to maximize their flexibility (in other words, limit their autonomy as little as possible). Accommodating flexibility proves quite challenging, especially to ensure that the agents behave correctly with respect to a protocol despite the flexibility allowed.

Commitments. Whereas the study of communications traditionally was based on mentalist concepts, the study of protocols has always been framed in social concepts. We use *commitments* as the canonical social concept for this purpose (Singh, 1999a). Although other such concepts can be defined, commitments are adequate for our present purposes. A commitment involves a debtor, a creditor, a condition or action, and a context. The basic idea is that the debtor is obliged to the creditor to bring about the stated condition or perform the action; usually even actions are modeled as conditions. The given commitment obtains within the context, which can refer to the organization such as a virtual enterprise or the state of North Carolina or even a trading abstraction such as a supply chain. Commitments can be manipulated by means such as being delegated (change the debtor), assigned (change the creditor), canceled (when the debtor would break a commitment), or released (when the creditor or context would release the debtor). Manipulations, especially cancellations, are obviously risky. They are constrained through further *metacommitments*, which are commitments that specify the circumstances under which base-level commitments can be manipulated. Importantly, adding metacommitments makes it possible to model many of the practical kinds of protocols in the applications of interest (Yolum and Singh, 2002).

4. Agent-Based Software Development

Broadly speaking, agent-based software development is about creating techniques and methodologies that exploit the key features of agents. The key fea-

tures of agents were introduced in the above discussion. Since there is some variability among these abstractions, there is also some variability among the programming abstractions that result from them. The other chapters in this collection provide a thorough treatment of the major such variations, so it would be superfluous (and, in any case, impossible) for us to include such discussions here. Instead, what we can do is to consider each of the key features of agents and study how they can be captured as computing abstractions, giving a representative means of doing so. Readers can then use the simplified discussions below as bases for framing the rich variations that are introduced in the subsequent chapters of this volume.

Autonomy. For practical purposes, the very use of an agent-based approach confers a certain level of autonomy on the participating components. The main methodological enhancements that result from autonomy are that the agents are proactive rather than merely reactive. They can initiate activities and thus participate in a richer variety of patterns of interaction than studied in conventional software engineering. As an example of this, a classical inventory of the major software engineering patterns includes over ten patterns (Shaw and Garlan, 1996), none of which adequately captures the proactivity of the participants. To be useful while being autonomous, agents should also be persistent, because otherwise a trivial way to get around an agent's autonomy would be to kill it and start another instance. Persistence means that the agents can participate in elaborate conversations and carry on long-lived transactions (more on this below).

Heterogeneity. As mentioned above, information heterogeneity is handled through conventional means such as ontologies. The deeper, process heterogeneity is constrained through a specification of the significant events that are exposed. The significant events along with so-called *skeletons*, which prescribe the allowed transitions on which the stated events occur. Some methodologies incorporate such skeletons and offer ways in which to create correct skeletons based on an examination of the desired interactions among the various agents (Huhns et al., 2002; Singh, 2000).

Dynamism. Dynamic configurability is one of the strengths of agent architectures and implementations. Agent approaches pioneered techniques variants of which are widely used, for example, in UDDI. However, the deeper challenge is not of discovery, which UDDI addresses, but of *selection*, that is, choosing an appropriate service implementation from among the several that may have been discovered. Some of the most relevant agents approaches, which are largely not yet incorporated into commercial approaches, involve

matchmaking. Well-engineered agent-based systems would usually involve a component for matchmaking. Recent approaches consider matchmaking based on semantic descriptions of Web services (Trastour et al., 2001). An alternative class of approaches is based on considering the quality of service that is obtained from a given implementation. A recent approach introduces a conceptual model for quality of service and embed it within an architecture where service selection is considered an integral component (Maximilien and Singh, 2002). Another recent approach enhances traditional recommender systems to handle service, rather than just product, selection (Sreenath and Singh, 2003).

Communications. The communications language is typically broken into three main components. The transport layer provides messaging, usually reliable to some limited level. The content layer provides a means for expressing the relevant domain details; it is typically associated with an ontology, which is specified outside of the communication language per se. Computationally, the greatest emphasis has been on the communicative acts. The communicative act layer specifies what the philosophers term the *attitude* of the communication, for example, whether the communication is an assertion, a directive, a promise, and so on. Depending on one's theory, this set of primitives can vary, but is typically a small number (under ten). Whereas it is usually clear what communicative act is required for what situation, practitioners have some difficulty with choosing the correct one in each case. Consequently, there is a tendency to choose just one communicate act (usually, this is an act corresponding to asserting a fact and is called "assert" or "inform") and load it with the meanings of all the other communicative acts. There have been some attempts at countering this tendency through rich models of dialogue and argumentation, with increasing success (Pasquier and Chaib-draa, 2003), although these approaches have yet to be expressed in hard-core deployed software methodologies.

Protocols. The specification of protocols has drawn a lot of attention. These approaches consider how the requirements for communication may be captured. One class of approaches emphasizes the communication aspect of protocols by beginning from rigidly specified interactions and systematically liberalizing them to accommodate additional behaviors. The method of choice is to capture the semantics of the protocols and reason about the semantics to enable alternative executions (Chopra and Singh, 2003).

Commitments. Some approaches combine these with the specification of significant events. The earlier work on commitments assumed that the desired commitments and metacommitments were determined directly by a designer based on an understanding of the interactions desired. Recent work has yielded more precise operational formulations of commitments, which can

then be used as a basis for building systems (Fornara and Colombetti, 2002). Temporal logic approaches now exist in which the semantics of commitments can be expressed and reusable patterns of interaction involving commitments can be formalized (Xing and Singh, 2003). These patterns can be used as a basis for designing a MAS with some guaranteed properties about the resulting interactions. Other recent work considers the methodological aspects of coming up with the right commitments based on an analysis of the desired interactions (Wan and Singh, 2003). This work is in the spirit of (Huhns et al., 2002; Singh, 2000) but geared toward commitments.

5. Critical Directions

The study of agents for software development has led to a vibrant body of work. The rest of this volume stands testament to the variety of technical challenges that have been explored, techniques that have been invented, methodologies that have been formulated, and systems that have been constructed. These approaches exercise to varying degrees the concepts introduced above.

However, open systems offer a larger set of challenges than have been adequately addressed in the current approaches. This is not a criticism of the existing approaches, because it would have been foolish to attack the deeper problems before the basic challenges had been resolved. Indeed, there is great value in being conservative so that one's approach has a better chance of success and thus a better chance of influencing the practical computer science community.

The success of the basic technologies for e-business, however, naturally draws attention to some outstanding challenges. At the same time, the credibility of the existing agents approaches suggests that agents approaches may have something to offer. In fact, we would go further and suggest that agents approaches are the only viable means for addressing these problems.

Pragmatics. We alluded above to the overarching problems of semantics and trust. There has been much progress in the formulation of languages and algorithms for capturing the semantics of information. But there is another side to the semantics, which we term *pragmatics*, which is not adequately handled by current approaches (Singh, 2002). In simple terms, pragmatics considers the usage of information, for example, within potentially elaborate contexts of interactions and business processes. Pragmatics is dynamic and, therefore, needs abstractions that can better accommodate changing contexts. Such abstractions are being developed, but they remain to be incorporated into software engineering techniques.

Trust. The challenge of trust is in some ways easier to motivate. In an open system, you have no control of the participants' behavior. So how can you trust

them (or those that are indeed trustworthy)? Current approaches for open architectures do not have a principled means for supporting trust. Traditional approaches have little to say except via support for representations such as certificate chains (Clarke et al., 2001). But, even in the best such approaches, certificate chains do not offer trust. They may offer a weak form of authentication, but the question of trust is left entirely to the user, who must decide how much trust to place in a given chain of credentials. The agents approaches have a natural advantage in developing refined notions of trust. Some of this work has a philosophical flavor, but computational approaches are emerging as well (Yu and Singh, 2002). Although these have yet to be mapped into practical methodologies, there is work underway to develop methodologies and thus to complete this aspect of the picture.

Monitoring and Compliance. Another major challenge for open systems is to determine whether the participants are complying with the applicable rules and policies. As remarked above, because the participants are autonomous, the only constraints we have on their behavior are the commitments into which they have entered. But how do we know that they are behaving according to their commitments? There is clearly a need for verifying compliance, for example, to ensure correct behavior. Trust does not replace compliance verification; instead methods for estimating trustworthiness depend on knowledge of whether a given party has complied with its commitments in the past. The problem of compliance is made harder by the fact that open systems are distributed and the information necessary to make the right evaluations may not be present at a given central site. In other words, monitoring interactions is difficult. An early work along these lines is (Venkatraman and Singh, 1999), but accommodating compliance checking and, more importantly, compliance verifiability into design methodologies remains an unresolved challenge. The basic idea would be to design MAS so that monitoring and compliance checking were an integral part of the design, thereby helping the participants develop increased trust in the system while it is deployed and used.

6. Conclusions

We presented a conceptual overview of the main abstractions associated with the concept of agency. We argued how the abstractions that correspond to MAS and to interactions among agents fit naturally with possible solutions to the challenges of open systems and showed how they are given a computational interpretation in modern modern methodologies and approaches for software engineering of agent-based systems.

Acknowledgments

The work described in this chapter was partially supported by the National Science Foundation through grants ITR-0081742 and DST-0139037. I have benefited greatly from discussions of these topics with several people, most notably, Mike Huhns, **Pinar Yolum**, Amit Chopra, and Ashok Mallya. I thank Franco Zambonelli for helpful comments on an earlier version of this chapter.

Chapter 2

ON THE USE OF AGENTS AS COMPONENTS OF SOFTWARE SYSTEMS

Federico Bergenti and Michael N. Huhns

Abstract Software agents are increasingly being used as building blocks of complex software systems. In this chapter we discuss the benefits and the drawbacks that a developer faces when choosing agents for the realization of a new system, instead of a more mature technology such as software components. In particular, we first compare agents to components and then highlight the differences and similarities engendered by the metaphors and abstractions that each provides. Then, we concentrate our comparison on reusability because of general agreement that reusability is one of the most important features to consider when adopting a development technology. We exploit agent-oriented concepts to define formally an asymptotic level of reusability, and we show how agents and components approximate it. The result of such a comparison is that agents are intrinsically more reusable than components.

1. Introduction

The creation and further development of AOSE in the last decade has promoted agents as a viable new way to develop complex software systems. AOSE gives to the developer all the flexibility and the expressive power of agents and it helps with the management of the software lifecycle in an attempt to improve the quality of the resultant software products.

During its short history, research on AOSE has undergone an important change of focus: initially it was meant only to provide methodologies and tools to build agent-based systems; today it is more concentrated on understanding the features that an agent-based approach can bring to the development of conventional software. This change of focus is not trivial and it corresponds to a radically different approach in adopting agents during the evolution of the software lifecycle. The first approach is based on choosing agents as the very basic abstraction of the development, before actually starting the software lifecycle. Such a decision is taken for reasons that fall outside of the software lifecycle

and is generally based on the nature of the system or on the complexity of the problem at hand.

The choice of agents as the very basic abstraction for development allows adopting the agent-oriented mindset for the entire lifecycle, from analysis of early requirements to the retirement of the system, as the Tropos methodology (Bresciani et al., 2001) suggests. The major drawback of this is that often there is no reasonable motivation for choosing agents to develop conventional systems, e.g., word processors and financial planning systems. This is the reason why a more modern approach to AOSE tends to move the decision on adopting agents after (or during) the phases for requirement analysis and requirement specification. The developer is not forced to envisage his/her system in terms of agents, rather he/she can concentrate on the requirements that will drive the subsequent design. The major disadvantage of this approach is that the developer may not exploit interesting features of agents, e.g., emergent behavior and generalization, because he/she has concentrated too much on the concrete requirements that come directly from the client.

Besides this drawback, the more recent approach has highlighted the problem of understanding when and how the developer should prefer agent technology instead of any more traditional, and possibly more mature, technology, such as object-oriented technology. In this chapter we address a particular aspect of this problem and we present the motivations for choosing agents instead of a technology that resembles agents from many points of view: software components (Szyperski, 1998).

In the following section we compare agents and components by taking into account five aspects that they share. Our results have maximal generality, because our analysis is not bound to a particular technology. In section 3 we extend the depth of our comparison by concentrating on reusability, which is one of the most important aspects of a development technology. We begin this by formally defining an asymptotic level of reusability, and then show how agents and components approximate it. The result of such a comparison is that agents are intrinsically more reusable than components. Finally, in section 4, we briefly discuss the implications of our comparison results.

2. Software Agents vs. Software Components

Since the first release of the FIPA specifications in 1997 (see <http://www.fipa.org>), researchers clearly understood the possibility of using agents as software components capable of exhibiting interesting characteristics, e.g., automatic reasoning and goal-directed behavior. In addition, FIPA chose to enable communication among agents by means of a CORBA interface, and this emphasized even more the strong interrelation between such abstractions. The

long (and sometimes pointless) debate on the differences between agents and objects, e.g., (Wooldridge, 2000), originated from this comparison.

Component-oriented software engineering proposes extensions of objects, e.g., Web Services (see <http://www.w3.org>), CORBA components (Suhail, 1998), JavaBeans (see <http://java.sun.com>), and .NET components (see <http://www.ecma-international.org>), as final a answer to the strong need of reusable building blocks that can be assembled to realize complete systems. Such components are interoperable across networks and (possibly) languages and operating systems, to give a developer maximal freedom in the deployment of a system. Nevertheless, the long-pursued dream of component-oriented software engineering does not end with the realization of a technology for reusable units of software, but it considers also the following ideas:

- 1 Commercial Off-The-Shelf (COTS) components, i.e., components that are available in a public market and that are assembled to create a value-added system. The quality and the cost of the system basically depend on the quality and costs of every single COTS component. Market forces should help in decreasing costs while increasing the quality of available components; and
- 2 Automatic assembly, i.e., the possibility of lowering the cost of the process of assembly of components through the use of automatic technologies. The quality and cost of the assembly process depend directly on the quality and cost of the available technologies for automatic assembly.

The use of COTS components combined with automatic assembly can lower the cost of a component-based system down to the direct investments related to each single component, summed to the cost of the technology for automatic assembly. Similarly, the quality of a system increases according to the quality of single components and of the technology for automatic assembly.

Our comparison between agents and components starts from the following table, where we show some important aspects of components and associate them with their agentized counterparts. More precisely, we consider the most important features of the agents' metamodel and compare them with the corresponding features of the components' metamodel. In order to give maximal generality to our results, we avoid considering any particular metamodels, e.g., the Microsoft .NET metamodel for components or the SMART framework (Luck and d'Inverno, 2001) for agents.

State Representation. Both agents and components are abstractions that comprise a state, but they have very different means to describe and to expose it to the outer world. The state of a component is represented though a set of attributes and a set of relations with other components. Attributes and relations can be public, i.e., other components can manipulate them directly.

Table 2.1. Features of the agents' metamodel and their component counterparts

Feature	Agent-oriented	Component-oriented
State	Mental attitudes	Attributes and relations
Communication	ACL	Metaobject protocol
Delegation of responsibility	Task and goal delegation	Task delegation
Interactions between parties	Capability descriptors	Interfaces
Interaction with the environment	New beliefs	Events

An agent has a mental state, i.e., its state is represented in terms of what it knows, e.g., its beliefs, and what it is currently pursuing, e.g., its intentions. The main differences between such models for representing the execution state are:

- 1 Agents cannot manipulate the state of another agent directly, but can affect it only through communication;
- 2 Agents have an explicit representation of their goals;
- 3 Agents have explicit knowledge of their environment, including other agents in the environment; and
- 4 Except for a unique identifier, agents do not have public attributes.

One of the main advantages of the agents' approach is that agents can use general-purpose reasoning techniques to support deduction and means-end reasoning. On the contrary, the attributes and relations of a component's state are not structured in a logic framework, so it is difficult to use general-purpose techniques; any deduction and planning process must be coded explicitly in the methods of the component.

Communication. The main difference between agents and components is in the mechanism they use to communicate. Agents use declarative Agent Communication Languages (ACLs), while components use metaobject protocols (Kiczales et al., 1991). In the agent-oriented approach, a message is sent only in an attempt to transfer part of the sender's mental state to the receiver. Let's take the FIPA ACL as an emblematic example: it defines performatives, i.e., semantic message types, together with feasibility preconditions, which must be true for the sender to send the message and rational effects, which are why the sender sent the message. When an agent receives a message, it can assert that the feasibility precondition holds for the sender and that the sender is trying to achieve the corresponding rational effect. This is basically a rather knotty way to let the receiver know that the sender wanted the receiver to know

that the feasibility precondition holds for it and that it is actually bringing about the rational effect. The advantage of using a structured ACL, instead of a more natural exchange of representations of goals, is that it simplifies the development of reactive agents capable of complex interactions. Reactive agents with no reasoning capabilities can exploit the performatives of the ACL as triggers that activate the state machine of the underlying interaction protocol. This is what JADE (Bellifemine et al., 2001) (see chapter 13) and similar platforms provide.

In the component-oriented approach, a message is sent for two reasons. The first is to directly manipulate the state of the receiver. This use of communications violates the autonomy of the component, which should be solely responsible for its own state. Most real-world technologies for implementing components prohibit direct manipulation of states, in an attempt to satisfy a software engineering goal of minimizing this sort of coupling among components. The second reason for sending a message is to force the receiver to execute the body of a method for the sender without explicitly communicating to the receiver why it is being forced to do so. The responsibility for such an execution is completely that of the sender: it is responsible for guaranteeing that preconditions hold and for causing any changes in the rest of the system that might arise during the complete execution of the method.

Delegation of Responsibility. As we have just pointed out, both for agents and for components the delegation of responsibility is based on communication and the differences in the way they delegate responsibilities justify the differences in their communication models. In the component-oriented model, the sender is solely responsible for the possible outcomes of a message: it does not need to say anything more to the receiver than “please do this under my responsibility.” Strictly speaking, components do not delegate responsibility to other components at all. In the agent-oriented model, the receiver is solely responsible for the outcome of its own actions and the sender needs to say also why it is requesting the service. A very important communicative act that an agent can perform is delegating one of its goals to another agent, e.g., through the FIPA achieve performative. This special communicative act, known as goal delegation (Castelfranchi, 1998), is the basic mechanism that agents use to delegate responsibilities.

The components’ metamodel does not comprise an abstraction of goal and components can only use task delegation. Components achieve their (implicit) goals by forcing other components to perform actions; agents might achieve their (explicit) goals by delegating them to other agents. This is the reason why it is common to refer to the agent-oriented communication model as *declarative message passing*: agents can tell other agents what they would like for them to do without explicitly stating how to do it. On the contrary, *imperative*

message passing is used for the component-oriented approach, because components cannot say to another component what to do without also saying how to do it.

The possibility of using only task delegation is a strong limitation for components, because goal delegation is a more general mechanism. First, task delegation is a special case of goal delegation: the delegated goal has the form *done(a)*, where *a* is an action, just like for the rational effect of the request performative in the FIPA ACL. Then, task delegation may inhibit optimizations. Consider, e.g., a component *S* with a goal *g* that needs component *R* to perform a_1 and a_2 to achieve it; *S* would ask to *R* to perform a_1 and then it would ask *R* to perform a_2 . As the two requests are not coupled though the underlying idea that *S* is trying to achieve *g*, *R* cannot exploit any possible cross-optimization between a_1 and a_2 .

If *S* and *R* were two agents instead of two components, *S* would simply delegate *g* to *R* and then *R* would decide autonomously the way to achieve it, i.e., it would decide how to perform a_1 and a_2 . This approach couples a_1 and a_2 through *g*, thus enabling *R* to perform cross-optimizations between a_1 and a_2 .

Interaction between Parties. The different communication models influence the way agents and components open themselves to the outer world. Components use interfaces to enumerate the services they provide and to tell clients how to get in contact with them. Sophisticated component models, e.g., (Meyer, 1997) equip interfaces with preconditions and postconditions.

The agent-oriented approach eliminates interfaces and provides agents with capability descriptors that depict what an agent can do, i.e., the possible outcomes of its actions, and how it can interact with other agents. The main difference between a capability descriptor and a postcondition is that the first can express how the state of the environment changes after the complete execution of an action. A postcondition can only assert how the state of the component changed after the action has been executed, because the environment is not part of the component's metamodel.

Interaction with the Environment. As we have just mentioned, an environment is a structural part of an agent's metamodel, while it is not part of a component's metamodel. Agents execute in an environment that they can use to acquire knowledge: agents are situated abstractions. Agents can measure the environment and they can receive events from it. In both cases, agents react to any change in the environment because of changes in their mental state. This is radically different from the component-oriented approach where the environment communicates with components only through reified events. Compo-

nents can react to an event only by constructing a relation with a reification of the event itself.

The component-oriented approach seems to better respect encapsulation than the agent-oriented approach: the state of the component is changed only when the component itself decides to change it in reaction to an event. If we consider this in more detail, we see that the agent-oriented approach also respects encapsulation. Agents have reasoning capabilities that are ultimately responsible for any change in their mental state. Any direct push of knowledge from an agent's sensors to its mental state is ruled through reasoning, and the mental state remains encapsulated.

3. Semantically Reusing Agents and Components

Since the beginning of computer science, reusability has been considered one of the main properties of a development technology. First procedures, and later classes were a direct response to the need for creating reusable units of software to, e.g.:

- 1 Speed-up the realization of new systems; and
- 2 Ensure the quality of systems that are realized though the composition of a number of readymade units.

Component-oriented software engineering has already explored most of the peculiarities related to building a system in terms of assembled components, and it identified three concepts that any technology meant to improve reusability should take into account: semantic interoperability (Heiler, 1995), semantic composability (Pratt et al., 1999) and semantic extensibility (Fankhauser et al., 1991). No formal definition for such concepts is available in the literature and the general feeling is that formalizing such ideas would require concepts that are not part of the components' metamodel. On the contrary, we can give formal definitions of such ideas by taking into account very basic elements of the agents' metamodel.

Semantic Interoperability. Previous research on software components has explored the problem of semantic interoperability in many ways, and the agent community has also begun investigating the subject. For example, the recent work on the characterization of the capabilities of Web Services (McIlraith and Martin, 2003) follows the lines of established results, e.g., (Jeng and Cheng, 1995). Strangely enough, there is no agreed upon definition for semantic interoperability and some variants of this concept are available in the literature with different names, even though this name has been in use for a while.

The idea of semantic interoperability comes from a reasonable extension of syntactic interoperability of components, i.e., the sort of interoperability

that CORBA and standards with similar aims (e.g., DCOM and Java RMI) provide. CORBA allows components to exchange messages and provides an agreed upon syntax for such messages. The semantics of the exchanged messages is implicit, i.e., the semantics of a call to a method of a CORBA interface is implicitly defined as follows: the call to the method actually causes the execution of the body of the method. Nothing is said on the concrete outcome of the call, i.e., what would happen to the world outside of the component that executed the body of the method after such an execution would be completed. This outcome is considered application specific and relies completely on the programmer, who is responsible for reading the documentation of the interface for deciding when and how to call the method.

Syntactic interoperability inhibits automatic assembly of components, because a client has no means to reason about the effects of a call it might have decided to perform on one of the methods of a service-provider component. Semantic interoperability is about extending the interface of a component with an explicit formalization of the outcome of a method call in order to allow a client to decide autonomously when and how to invoke that method.

What we have just described can be applied to agents if we concur that invoking a method on a component is somehow similar to asking an agent to perform an action. Exploiting the characteristics of agents, we can formalize semantic interoperability as follows:

Definition (Semantic Interoperability, Client Standpoint). Given two agents C and $S \in \text{acquaintance}_C$, they are said to be semantically interoperable if and only if:

$$\forall g : G_Cg, G_{Cdone}(\text{delegate_to}(C, S, g)) \Rightarrow K_S G_{Cdone}(\text{delegate_to}(C, S, g))$$

where $\text{delegate_to}(C, S, g)$ is a sort of abstract action of C whose outcome is: $K_C G_S g$.

This definition states that if (at some point in time) an agent C wants to achieve g , and it wants to delegate such a goal to S , then S will know of such a desire. In this way we can easily capture the lack of information loss, which is the core of semantic interoperability: if an agent has a goal, then it can transmit that goal to a service-provider agent without any loss of precision. It does not really matter how the goal is communicated, the only important result of the communication is the delegation of the goal to the service provider.

This definition of semantic interoperability takes the client standpoint, because C is the originator of g and nothing is said about S wanting to provide its services to a set of possible clients. A similar definition is trivially possible taking the server standpoint, but such a definition is basically equivalent to the one we showed and its discussion would not add much to the aims of this chapter.

A fairly interesting consequence comes from this definition of semantic interoperability: if we consider a MAS where agents are only intended to interoperate semantically, then a very basic ACL with the *achieve* performative only is sufficient. This is not strange at all, because it easily generalizes the available work on ACLs, as mentioned in the previous section.

The achievement of semantic interoperability is not only a way for improving reusability, it is also a possible way for promoting optimization. With everyday syntactic interoperability, agents achieve their goals by possibly asking other agents to perform actions, i.e., exploiting task delegation towards other agents in an attempt to achieve their goals. Semantic interoperability exploits goal delegation and this may promote cross-optimization, as discussed briefly in the previous section.

Semantic Composability. The assembly of agents to realize a MAS is not only a matter of making agents communicate in the best way, but also allowing them to find each other. Interoperability is necessary, but not sufficient, for composability: semantic interoperability requires $S \in \text{acquaintance}_C$ and we need to elaborate on this to achieve full semantic composability.

Semantic composability has been studied for a long time in the literature of component-oriented software engineering, starting from well-known results on the composability of objects obtained by researchers who are now active in the community of aspect-oriented programming (Aksit et al., 1993). The basic idea behind semantic composability is that a component should be free to compose the services provided by a set of service-provider components with no constraints deriving from locating the right service providers or from possible mismatches between the interfaces of such service providers. It requires that the things being composed not only have compatible interfaces, but also make consistent assumptions about the world. Semantic composability has already been extended to agents (Sycara et al., 2002), and we can make it more formal by exploiting the same technique that we used for semantic interoperability. We can say that two agents are semantically composable if no constraint is imposed on the way agents delegate goals and, more formally, we can define semantic composability as follows:

Definition (Semantic Composability). Given a set of n agents $MAS = \{A_1, A_2, \dots, A_n\}$, they are said to be semantically composable if and only if:

$$\forall C \in MAS, \forall g : G_C g, \exists A \in MAS : \text{solves}_A(g),$$

$$G_C \text{done}(\text{delegate}(C, g)) \Rightarrow \exists! S \in MAS : \text{solves}_S(g), K_S G_C \text{done}(\text{delegate}(C, S, g))$$

where solves_A are the goals that A can solve and $\text{delegate}(C, g)$ is a sort of abstract action of C whose outcome is: $K_C(\exists X \in MAS : G_X g)$.

This definition states that if an agent C has a goal and there is an agent S available in the MAS capable of achieving such a goal, then C can delegate the

goal to S with no loss of precision caused by communication. In this way we can capture the lack of information loss that semantic interoperability entails, without the need of requiring C to know S and to desire to delegate its goal to that S . It does not really matter how or to whom the goal is communicated, the ultimate result of the composition is that an agent of the MAS would achieve the goal for C .

This definition does not require the client to know the service-provider agent prior to the delegation, and it does not guarantee that the chosen provider would be known after the delegation. This is compatible with the common approach of explicitly choosing the service provider, because the two approaches are both captured by the definition: the client can identify the service provider of choice in its goal. For example, if an agent C wants S to achieve goal j for it, then the goal that C is bringing about is actually $g = K_S G_C G_{Sj}$.

Semantic Extensibility. Taking the literature on component-oriented software engineering into account, we see that reusability is pursued not only by means of composing reusable components, but also by making such components extensible (Booch, 1994). Extensibility provides mainly two possibilities of reuse:

- 1 Implementation of new components as extensions of available components; and
- 2 Substitution of an existing component with a different one with (possibly) no changes in the rest of the system.

The first approach is traditionally considered the base of object-oriented programming: it supports the creation of new classes of objects by means of inheritance and polymorphism. This is still a good way to bring about reusability, but nowadays the second approach is preferred because it allows reusing entire systems and not only single classes. This so-called *framework-based* reusability relies on the possibility of substituting a component with another component without the rest of the system (i.e., the framework) being aware of such a substitution.

Object-oriented and component-oriented paradigms achieve such a framework-based reusability by means of inheritance and polymorphism, because they assume that if two components belong to the same class, i.e., they are of the same type, then they are substitutable. This is obviously not enough and some extensions to such an approach have been already proposed (Meyer, 1997). In particular, the main problem of approximating substitutability with type equivalence is that two classes may provide the same methods, but the semantics of such methods, i.e., what they do on the world outside of the component, may be completely different. In other words, two classes may be structurally identical, but semantically different (Fankhauser et al., 1991).

The idea behind semantic extensibility is that we want to have the possibility of substituting a component with another component extending the features provided by the first component, while preserving the semantics of the operations that clients were able to perform before the substitution.

Taking the agent-oriented mindset and exploiting the formalisms that we have introduced previously in this section, we can formally define semantic extensibility as follows:

Definition (Semantic Extensibility). Given two agents B and D , we can say that D is a semantic extension of B if and only if:

$$\forall g : \text{solves}_B(g) \Rightarrow \text{solves}_D(g)$$

This definition states that (at each point in time), what B can solve is also solved by D , i.e., from the point of view of any possible client interested in the services that B may provide, they are substitutable.

Semantic extensibility together with semantic composability maximizes the reusability of agents, at least if we adopt the assumption of considering agents as the atomic units of reuse. Agents are composed freely on the basis of their goals and they can be substituted with other agents with extended capabilities with a complete reuse of the MAS surrounding the substituted agents.

Approximating Semantic Reusability. The model of reusability that we have just discussed is obviously idealistic, because it does not provide any operational means for supporting composability and extensibility. Nevertheless, if we make goals_A and solves_A public and explicit, then semantic composability is just a matter of passing a goal from a client to a service provider and, in the most general case, it is just a matter of communication. We could exploit a matchmaker agent capable of connecting a client with a service provider, or we might rely on the middleware infrastructure. In this last case, e.g., we could exploit a tuple space forwarding goals from clients to service providers, or we could rely on a direct message passing that the programmer coded explicitly in the program of the client. Similarly, making solves_A public and explicit guarantees semantic extensibility, because a client can always check solves_S before requesting a service from a service provider S .

Unfortunately, goals_A and solves_A cannot be computed in the most general case and we can only rely on public and explicit approximations of them. Components and agents provide different approximations and the advantages that agents have over components in terms of reusability derive from these different approximations.

The ParADE framework (Bergenti and Poggi, 2001) was designed to maximize the interoperability of agents and it approximates semantic reusability as follows:

- 1 $B_A \simeq K_A$: the knowledge of an agent is approximated with what the agent believes, i.e., what it has deduced by applying steady rules to its measurements of the environment;
- 2 $I_A \simeq G_A$: the goals of an agent are approximated with the intentions that it calculates from its beliefs and from the rules that drive its planning engine; and
- 3 $capabilities_A \simeq solves_A$: the goals an agent can solve are approximated with the post-conditions of its feasible actions. These postconditions take into account the state of the agent and of the environment after the complete execution of an action.

The components' metamodel relies on even stronger assumptions:

- 1 $state_A \simeq K_A$: the knowledge of the component is approximated with the state of the component, i.e., the values of its attributes and its relationships with other components;
- 2 $postcondition-of-next-call_A \simeq goals_A$: the goals of a component are approximated with a singleton set that contains the postcondition of the method that the component is about to invoke; and
- 3 $postconditions_A \simeq solves_A$: the goals a component can solve are approximated with the postconditions of its methods. Such postconditions are defined on the state of the component after the complete execution of a method and nothing is said about the state of the environment.

Roughly, agents approximate semantic reusability better than components, because the element of the architecture that is in charge of enabling the flow of information between a client and a service provider, e.g., the matchmaker agent, has more precise information to perform its job.

Agents approximate semantic extensibility better than components because the capability descriptors that they use comprise conditions on the environment surrounding the agents, while the postconditions of the methods of components consider only the state of a service provider after the complete execution of a method. Therefore, agents can give very precise information on the outcome of an action for the purpose of guaranteeing semantic extensibility.

4. Discussion

Agents not only are suited for uncommon types of applications where the advanced characteristics of agents, such as learning and autonomy, are required, but also represent a valid alternative to other solid technologies because agents:

- 1 Provide the developer with higher level abstractions than any other technology available today (Bergenti, 2003); and
- 2 Have concrete advantages over components in terms of reusability.

The first point, i.e., working with higher level abstractions, has well-known advantages, but it also has a common drawback: slower speed of execution. In order to fully exploit the possibilities of agents, we need to implement an agent model with some reasoning capabilities and agents of this sort are likely to be slow. Nowadays, this does not seem a blocking issue because speed is not always the topmost priority, e.g., time-to-market and overall quality are often more important. As far as the second point, reusability, is concerned, the improvement that agents obtain comes at a cost: slower speed again. The use of goal delegation instead of task delegation requires, by definition, means-end reasoning and we face the reasonable possibility of implementing slow agents.

Fortunately, in both cases the performances of agents degrade gracefully. We can choose how much reasoning, i.e., how much loss of speed, we want for each and every agent. In particular, we may use reasoning for agents that:

- 1 Are particularly complex and could benefit from higher level abstractions; and
- 2 We want to extend and compose freely in many different projects.

On the contrary, we can rely on reactive agents, or components, when we have an urge for speed. This decision criterion seems sound, because the more complex and value-added an agent is, the more we want to reuse it and compose it with other agents. Moreover, reactive agents are perfectly equivalent to components and we do not lose anything using the agent-oriented approach instead of the component-oriented approach.

Chapter 3

A SURVEY ON AGENT-ORIENTED ORIENTED SOFTWARE ENGINEERING RESEARCH

Jorge J. Gómez-Sanz, Marie-Pierre Gervais and Gerhard Weiss

Abstract This chapter presents a selection of current research works on agent technology which are focused on the development of MAS. The purpose of this chapter is to guide developers through existing theories, methods, and software that can be applied in each stage of a development. However, this guide is not exhaustive due the amount of agent-related research works. Thus authors have added references to consult other information sources and complement the information given here. Readers are encouraged to consult these external references in order to obtain a more accurate view of the field.

1. Introduction

This past decade, developing a MAS has evolved from an art to a structured discipline. Existing results in MAS research enable a developer to construct MAS easier than before. Among others, there are tools that can produce complete MAS from a specification, libraries of components that deal with concrete MAS issues (distributed planning, reasoning, learning), and theories that describe MAS behavior and properties. Knowing all of them requires a great effort. Existing surveys facilitate this task, but it is hard to give an overall view of what software, theories, and methodologies exist, and how they are applied to MAS development.

To alleviate this problem, and make the information easier to apprehend, authors of this chapter have structured existing references into sections that deal with MAS development from an engineering point of view. Thus, there are sections that consider *analysis*, *design*, *implementation*, and *testing*. The purpose is to make this chapter less a survey and more a manual for MAS development. This way, developers with some background on conventional *Software Engineering* (SE) will see how they would do with agents what they do using other paradigms. Also, this approach benefits beginners and specialized re-

searchers in this field by introducing a general picture of the development, not just solutions to concrete problems.

The criteria to distribute agent research into these stages have been obtained from the SE itself. In SE, each stage of a development process deals with concrete topic and pursues the production of certain documentation, specification or software. Trying to adapt topics considered in MAS research and the topics associated to each development stage, the authors of this chapter have prepared a brief summary of what readers can find along the chapter:

- **Analysis.** This section deals with research work in agents helping to obtain a problem description. In traditional software engineering, this relates to the expression of *requirements*. In the agent domain, there are agent approaches that deal directly with these *requirements*, but there are also other hybrid approaches that have been adapted to the agent concepts. This section includes considerations about the role of agents in analysis purposes, specification languages that use agent concepts, and support tools for these languages.
- **Design.** This section considers how to facilitate a design of a MAS using agent concepts and technology. In software engineering, design covers the study of how to realize analysis elements into another specification – software architecture, components, expected behaviors – that can be directly implemented. To translate analysis specifications, it is necessary to know how to build agents from scratch and using agent development environments.
- **Implementation.** This section surveys different approaches to use agent technology in realizing concrete agent oriented designs. It proposes several agent oriented languages, software libraries, frameworks, and support tools.
- **Testing.** This section is a review of testing methods based on agent concepts enabling to check that a MAS satisfies initial requirements and that it has been built with no errors.

It should be remarked that the results mentioned in these sections are focused in the AOSE domain. There is no intention to evaluate state of art of SE. In principle, existing formalisms, techniques and methods (e.g., object-oriented ones) can be applied to develop agent-oriented software. However, a problem with these approaches is that they fail, or tend to fail to capture the essence of agent orientation (Jennings, 2000). Consequently, approaches devoted to the agent orientation are needed and such approaches are surveyed in this chapter. For the sake of accuracy, the chapter uses software engineering terminology when possible. This terminology is found in (Pressman, 1982; Sommerville, 2001).

Each section contains recommendations of the authors of this survey about specialized literature and software. Though authors of this chapter have tried not to forget any relevant research, reading this chapter is not sufficient to obtain an adequate knowledge of the area. So, it is strongly recommended to read the papers referred in this work, as well as subscribing to organizations like AgentLink or AgentCities in order to be up to date with new results from research work.

2. Analysis

The analysis bases on the obtention of requirements in order to derive a description of what has to be built. A requirement determines what a software system should do and defines constraints on its operations and implementation. According to (Sommerville, 2001), there are several types of requirements. Among others, there are user requirements, system requirements, software, functional, and non-functional requirements. The process for obtaining requirements is not trivial. In fact, there is a discipline, named *Requirements Engineering* (RE), with this purpose. RE will be reviewed later in this section.

Agent research centers in functional and non-functional requirement definition. Functional requirements are statements of the services that the system must provide, or descriptions of how some computations must be carried out. Non-functional requirements are product requirements which constrain the system being developed, process requirements which apply to the development process and external requirements (Sommerville, 2001).

Despite the type of requirement, gathering them does not imply the choice of an agent model, since most of existing approaches make general assumptions about agents. A model of agent determines what elements are required to specify agents and establishes a predefined behavior. A comprehensive collection of articles, and a survey of features of existent agent models, can be found in (Huhns and Singh, 1998). Choosing an agent model at this point may couple too early a development with different agent architectures. So it may be better, unless it is a requirement, to leave this choice to the design stage.

Requirements themselves are understood in different ways by researchers in agents. Existing results can be categorized in one of these:

- Agent approaches oriented towards requirements representation. These representations include the concept *requirement* as first class citizen. These works are referred in the RE literature as well.
- Agent approaches that represent the system to be built using formal methods. These methods reuse formalisms tested in SE. They are directed towards the obtention of a formal specification.

- Agent approaches that represent the system to be built using diagrams. A frequent solution in SE is the use of diagrams to represent different aspects of a system. Agents have followed this line proposing different types of diagrams and concepts to capture the internals and externals of MAS. The specification obtained does not need to be formal.

Developers can decide at this moment what kind of analysis is more adequate to their situation. To present research works in each category, the chapter have been divided into three subsections.

2.1 Agent Approaches Specialized in Defining Requirements

This section studies the role of agents in RE. Applying agents in RE implies what some authors name *Agent Oriented Requirement Engineering* (AORE). These approaches are characterized by ascribing a more important role to the *agents*. However, according to surveys like (van Lamsweerde, 2000), the *goal* concept is more extended than *agent* concept to represent requirements. As this may disorient non-experienced developers, it is suggested that they read (Yu and Mylopoulos, 1998), which includes an extensive argumentation in favor of using *goal* as a first class concept when determining requirements. To have a pure, i.e., a non agent oriented view of requirements engineering, readers can consult (Zave, 1997) or (Nuseibeh and Easterbrook, 2000). The main difference of these works with respect to (van Lamsweerde, 2000) is that the *agent* concept is not so important. (Zave, 1997) categorizes requirements engineering works taking into account the kinds of problems that are addressed, and types of solutions to these problems. (Nuseibeh and Easterbrook, 2000) is a detailed review of RE that introduces different stages identified in RE processes and related works for each one.

Integrating agents and RE is not easy. For instance, (Yu, 1997b) indicates that *agent* is a concept to be delineated with different requirements like *being distributed*, *being intelligent*, or *being autonomous*. The problem is that the *agent* concept also ties in with concrete implementations that already carry their requirements. So, to make sense an agent-oriented approach in RE, it is better to use initially more abstract concepts like *role* or *actor* and once what is required from an agent is clear, assign roles to agents. Later on, agents will be implemented so that initial requirements hold.

One of the main references in AORE is i^* (Yu, 1997a). It is a framework that uses actors, beliefs, commitments, and goals to model organizational environments and their information systems. There are examples of using i^* in (Yu et al., 2001), that indicates how to identify the importance of a piece of information in a organization, and (Yu, 1999), that models a real furniture company. More examples can be found in the home page of the author

(see <http://www.cs.toronto.edu/~eric>). i^* has been also the starting point for other frameworks, concretely *Non-Functional Requirements* (NFR) (Chung et al., 2000) and *Goal-oriented Requirement Language* (GRL) (Yu and Liu, 2002). There are examples and tutorials available for GRL at <http://www.cs.toronto.edu/km/GRL>. i^* has a support tool called *Organization Modelling Environment* (OME) (Yu and Liu, 2003). This tool can be downloaded if previously a license agreement is sent to authors. OME also supports other notations, concretely GRL and NFR. So the same tool gives support for slightly different RE approaches.

Recently, i^* has been adopted as the underlying framework for an AOSE methodology named Tropos (Mylopoulos and Castro, 2000) (see chapter 5). Tropos has added to i^* a development process and automated translation methods from a i^* specification to agents based on JACK agent platform (Busetta et al., 1998).

Besides i^* , there are also two other classic works like KAOS methodology (Dardenne et al., 1993) and Albert (Dubois et al., 1994). KAOS also considers agents, actions, entities consumed by actions, and other relationships among entities. Agents are responsible for the execution of tasks in order to satisfy a goal, i.e., a requirement. An example of how to apply KAOS is (van Lamsweerde and Massonet, 1995), a paper that describes a distributed meeting scheduler. Notation of KAOS appears in (Dardenne et al., 1993). KAOS has also its support tool, which is named GRAIL (Darimont et al., 1997).

Albert is a pure specification language with more detailed semantics than KAOS or i^* semantics. In fact, it is less ambiguous than other approaches reviewed here. So its presentation has been moved to the next section.

Previous approaches lack of elements to model requirements with respect to agent interaction. In this sense, a recurrent solution is *role modeling*. It is oriented towards functional requirements, expressed with tasks, services, and roles integrated in workflows. Research from Kendall (Kendall, 1998) is a classic in this line of research. There are studies in identifying roles (Kendall, 1998) inside a development process though most of the work is done at the analysis level. This work has had a big influence on other works, such as Zeus (Nwana et al., 1999) or MaSE (DeLoach, 2001) (see chapter 6). Kendall's ideas constitute the method applied in the initial steps of Zeus (Collis and Ndumu, 1999) and MaSE (Wood and DeLoach, 2001) methodologies. For a more object-oriented approach, readers can consult (Depke et al., 2001), that considers functional requirements as UML use cases where roles participate in performing certain tasks. ODAC is also a methodology adapting standards from distributed object computing to agent and using UML (Gervais, 2003). More details on approaches using UML are given in the *UML based approaches* section.

2.2 Analysis Applying Formal Methods

Formal methods are mathematical modelling techniques applied to a software engineering process in order to obtain a non-ambiguous correct specification of the system to be built (Pressman, 1982). The expected output of a formal method is a formal specification. This specification can be used with different purposes, besides formal verification, as it is mentioned in the tests section. As (Wooldridge, 1997) remarks, a formal specification can be compiled to executable code or interpreted directly, without user intervention.

There are several research works that show how to apply formal methods in generating a specification of a MAS.

Abstract State Machines (Gurevich, 1984) are the formalism employed in MAS-CommonKADS (Iglesias et al., 1998b) to represent behavior of the system at a high level. Concretely, MAS-CommonKADS uses *Specification and Description Language* (SDL) (ITU, 1999), a standard language used to describe systems in the telecommunications domain. However, the degree of details in SDL is quite high. This formalism is complemented with Class Responsibility Collaborator (CRC) cards as a representation method to gather information about different aspects of the system, like tasks specification or the purpose of a system service. CRC cards are templates with slots that developers must fill using a concrete language. Other works reuse state machines, like agentTool (DeLoach, 2001) that uses them to represent the behavior of internal components of agents and protocols as well. This solution shares with SDL a common view of communication as interchanged messages among different state machines. In this line of research, developers can consult (Rosenschein and Kaelbling, 1995), that show how to translate first order logic formulae to state machines using automated methods. Though it is not exactly the same, Petri Nets have been used many times to express the behavior of a component. As an example to its application to the MAS domain, readers can consult (Xu et al., 2002) and (Demazeau, 1995). The first directly models the internal behavior of agents with this formalism. The second introduces Petri Nets as an abstraction to implement agent synchronization.

Z (Spivey, 1992) is a mathematical formalism based on sets manipulation. There have been experiments in adapting Z to the agent domain (d’Inverno and Luck, 1996; d’Inverno et al., 2000). Benefits of using Z are mainly reusing the big amount of software and tools available to this language, what includes automated code generation and formal verification methods.

Logics appear frequently as formalism to represent agent behavior, perhaps due to the the extensive use of logics in classic artificial intelligence. For a review of logics in a MAS context, readers can consult (Singh, 1997). A short introduction of modal logic to MAS appears in (van der Hoek, 2001) and (Wooldridge and Jennings, 1995a). According to these reviews, modal

logics are playing a main role in the agent research field. This is a novelty, since AI centers on propositional logic to represent knowledge and reasoning (Genesereth and Nilsson, 1987) with few attention to modal logic. In the agent domain, modal logic can handle formulas that do not satisfy *extensionality*. Extensionality says that, to determine the truth-value of a formula, only the truth-value of its subformulas must be considered. Due this property, time and desire cannot be modelled in classic logic, see (van der Hoek, 2001) for further explanations. Main works in modal logics in this field are the formalization of BDI (Bratman, 1987) model in (Rao and Georgeff, 1991) and intentional logic from (Cohen and Levesque, 1990). The first one was extended with architectures and methodologies (Kinny et al., 1996) and is referred in most works in the field. The second remarks the role of intentions using *intentional logics*. It centers on how beliefs, desires, intentions relate to agent actions. As a detailed example of a framework with BDI formalization and tool support, readers should review DESIRE (Brazier et al., 1994; Brazier et al., 1997), framework for *DEsign and Specification of Interacting REasoning components*. Another relevant work in temporal logics, which are modal logics, is Concurrent-METATEM (Fisher, 1994), though it relates to agent implementation rather than specification. However, it is interesting to read (Fisher, 1995) since it sketches how to apply this logics in a software process as a whole. Though it does not focus in the whole process, Albert (Dubois et al., 1994) proposes a variant of temporal logic with extensions to consider actions, agents, and constraints of the behavior of the system. This language, mentioned in the previous section, was designed to gather requirements that later could be processed using formal tools, like theorem proving software. The Web site of Albert is <http://www.info.fundp.ac.be/aibert> to know more details or contact authors.

Gaia (see chapter 4) also applies logics to describe some aspects of the system. Like MAS-CommonKADS, uses set of CRC-like cards whose slots are filled with logic formulae. It is not completely formal because these logic formulae, in the available examples, are not tied to any model or implementation, so their interpretation depends on the developer. However, these can be considered as a good approach to integrate SE methods together with formal ones.

Situation calculus (McCarthy and Hayes, 1981) is another formalism applied to model MAS. Situation Calculus is a first-order language (with some second-order features) for representing dynamic domains. In the agent domain, a key work is ConGOLOG (De Giacomo et al., 2000), a concurrent language to generate computational models that constraints task execution according to their preconditions and side-effects. The computational model that it defines executes tasks if and only if their execution will not take the system to instability. Though computationally it is an expensive problem, they have relaxed ConGOLOG constrains so that costs are affordable.

2.3 Analysis using Diagrams

Some methodologies represent part of the analysis results using diagrams. Diagram based representations may not be formal. Some approaches propose diagrams that are interpreted by a developer, like UML. In those approaches, different developers can derive different interpretations. However, there are also diagrams that are not ambiguous at all, like Petri-Net graphic representations of a protocol or Entity-Relationship diagrams to represent databases tables. This section reviews two kinds of diagram based approaches. The first deals with UML applied to the agent domain. The second with meta-modelling languages as specification language of the MAS at the analysis level.

UML-Based Approaches. One the most widespread approaches of this kind is AUML (Bauer et al., 2001) (see chapter 12), a project aiming at bringing agent concepts into UML (OMG, 2000c). As a suggestion, readers should start by (Odell et al., 2000) since it is one of the first papers about AUML. One of the relevant results of AUML is protocol diagrams notation, which is being considered as a new notation for the standard UML to express concurrence and decision. AUML Web site is <http://www.auml.org>. It contains working documents and articles about applying AUML to model different aspects of a MAS (Bauer et al., 2001; Odell et al., 2001). Unfortunately, there are no support tools for AUML.

PASSI stands for *A Process for Agents Societies Specification and Implementation* (Burrafato and Cossentino, 2002) (see <http://www.csai.unipa.it/passi>), and it is a recent work that reuses UML tools as front-end. It applies a UML representation of elements belonging to an architecture for a better understanding and handling. Concretely, Rational Rose is supplemented with a customized plugin that provides PASSI extra functionality. As an example of PASSI modelling, readers can consult (Burrafato and Cossentino, 2002) that shows modelling of a book store company. In this line of research, something similar has been proposed in (Bergenti and Poggi, 2001). It is a framework and a programming language that facilitates the definition of planning capabilities of agents. This approach inputs an XMI (OMG, 2003a) description of UML diagrams to generate code directly to a target agent platform. XMI is a model driven XML Integration framework for defining, interchanging, manipulating and integrating XML data and objects. As it is a standard (issued by OMG), any UML compliant tool should be valid as front-end.

Though these research works are very complete, they do not address properly MAS definition using a development process. Actually, they propose a too simple development process. Those readers interested in a more detailed solution to perform analysis, may read ADELFE (Bernon et al., 2002) (see chapter 8). It proposes a very detailed analysis method reusing UML notation

that can be adapted to several agent models. The point with this work is that it offers a general view of what elements are implied in a MAS generation, which is complementary of (Wooldridge, 1997).

Meta-Modeling as Specification Language. Meta-modelling is a technique for model description. Meta-modelling consists in describing types of objects, their properties, relationships and how they appear together in a model. This description is called a meta-model. A model is the instance of a meta-model and it conforms to a set of constraints defined in the meta-model, like *among objects of type A and B there can be only relationships one to one of type C*. Readers interested in knowing more about meta-modelling can consult (OMG, 2000b; OMG, 2000c), where a meta-modelling language is presented and applied to describe application data, program interfaces, or diagrams.

Why using meta-models in the agent domain? Citing (Gomez-Sanz et al., 2002), meta-models are useful as a kind of templates for generating agent models. They describe what any MAS should have. With meta-models of a MAS, the mission of the engineer turns into instantiating these meta-models in order to define the entities that may appear in a concrete MAS.

Meta-models have been used to specify AORE notation, like in (Dardenne et al., 1993), where KAOS elements were specified using a meta-model. Another referred work in the MAS domain is the AAlaading framework or AGR (Agent-Group-Role) (Ferber and Gutknecht, 1998). In that paper, authors show how to model organizations of agents using meta-models. That research has evolved into the MADKIT platform (see <http://www.madkit.org>). Examples of how this meta-model integrates with MADKIT are available in the literature, e.g., (Gutknecht and Ferber, 2001). MESSAGE (Caire et al., 2001b) (see chapter 9) proposes a meta-model for MAS that bases on engineering practices. As it is presented in detail in this book, it will be enough to say that its meta-model can be accessed at <http://www.eurescom.de>, where there are also prototypes and examples of specifications. Following the steps of MESSAGE, readers may also consult INGENIAS (Gomez-Sanz and Pavon, 2003). INGENIAS official Web site is <http://ingenias.sourceforge.net>. It also provides examples and Java-based development tools. Finally, meta-modelling is applied too in (Knublauch and Rose, 2002). (Knublauch and Rose, 2002) presents a visual modelling tool, AGILShell, and a notation to specify MAS. According to authors, this notation is specially suited for the design of agents that support the information flow between humans in existing work groups.

3. Design

According to conventions in software engineering (Pressman, 1982), analysis refers to what has to be done, whereas design determines how it could be done. Design also supplements analysis with information that deals with

implementation choice. So design aims at producing concrete instructions that allow programmers or tools to generate a system that satisfies analysis requirements.

Having selected a concrete analysis method influences the kind of design to be performed. Clearly, some of the works introduced in the previous section already include design concerns. So it would seem natural to continue the design using the same method. However, analysis does not compromise that all. Certainly, an experienced developer can choose any of the previous analysis approaches and still have freedom to selected another different kind of design. The link between design and implementation is harder to broke.

This section introduces two main tendencies in an agent oriented design. Some works suggest that the design activity mainly consists in a refinement of diagrams (Collis and Ndumu, 1999; DeLoach, 2001). These works relay on a specific *development environment* that translates diagrams to code. A development environment is a tool or set of tools capable of performing tasks required by a developer in order to build a software system. Usually, a development environment also defines a set of components, or a framework, to be used in the final system. This reduces the set of decisions to be taken during design. At the same time, it also makes the design less flexible, because there are parts that cannot be changed easily. Another trend is not to obviate these decisions and face the whole development from scratch, selecting adequate frameworks and libraries. (Bernon et al., 2002; Kinny et al., 1996) belong to this kind of works. Indeed, this is more flexible, but also harder to realize since it requires quite more experience than the other alternative.

3.1 Design with an Development Environment

Development environments for MAS building are oriented towards rapid prototyping. First development environments for MAS generation combined a graphical front-end and a MAS framework. The purpose of this front-end was to facilitate the MAS framework configuration. The resulting prototype was an instantiation of this framework. These environments for MAS generation were Zeus (Nwana et al., 1999) and AgentBuilder (see <http://www.agentbuilder.com>). Zeus includes ontologies, communication, and planning. It is a really good tool if a developer wants to create MAS quickly and experiment with their features. AgentBuilder is supposed to be the evolution of the agent programming language Agent0 (Shoham, 1993). Semantics of the models bases on the original Agent0 programming language. As Zeus, it includes a visual editor, simulator, and debugger. Unlike Zeus, it gives developers more control of the development environment, enabling developers to add new customized modules, called *Project Accesory Classes* (PACs), for connecting AgentBuilder agents with legacy applications.

These environments are adequate for rapid prototyping. Nevertheless, there are two risks on using them. First, this kind of tools may accelerate the development, but there is no guarantee that the resulting prototype will satisfy initial requirements. The problem may not appear at the beginning, but later on, when development needs become more strict. Besides, underlying frameworks of these tools cannot be modified easily. Usually, their code is not documented, so changing one component is risky. It may cause a general crash of the whole application. Second, developers depend on the authors of the tools to have updated versions with less bugs or improved functionality. By applying the tool to different domains, bugs are likely to appear. Testing of these tools cannot be exhaustive enough to ensure bug-free applications, specially when they are not commercial. Therefore, having regular updates of these environments is fundamental. The conclusion is that a rapid prototyping tool may not be the solution to all kinds of developments, specially if there is few budget available to buy good commercial tools, such as AgentBuilder.

Then, should rapid prototyping be discarded? The answer is no. There are other tools that propose a different way of generating prototypes. The proposal is to decouple the development environment and the predefined prototype. These tools support analysis and design and can produce code into any language. This is the case of agentTool (see <http://www.cis.ksu.edu/~sdeloach>, and chapter 6), PASSI (Burrafato and Cossentino, 2002) and INGENIAS IDE (see <http://ingenias.sourceforge.net>).

The two first act as translators of the specification language they use to a concrete implementation language. In the design, agentTool uses state machines notation, UML sequence diagrams and some variants of class diagrams to represent internal components of agents and the agents themselves. As a meaningful feature of agentTool, it uses a protocol verification tool, named SPIN (Holzmann, 1991), that prevents deadlocks. INGENIAS IDE, in the line of MESSAGE, uses a language built of common elements in MAS specifications (agents, tasks, resources, organizations, etc.) and a hierarchy of relationships that may appear among them. The result is different from UML, though it is constructed in a similar way. In a INGENIAS design, a developer centers on taking analysis elements and enhances the specification with more details and diagrams that illustrate how it could be done. Like UML, INGENIAS IDE uses incremental development techniques.

The two last, PASSI and (Bergenti and Poggi, 2001), propose reusing existing UML design tools and complementing them with agent code generation facilities. These tools also decouple the graphical front-end from facilities to translate a graphical notation to pieces of low level code. Later, these pieces are put together with other existing pieces of low level code. In the case of (Bergenti and Poggi, 2001) documentation is available. However, PASSI does not provide enough information about its approach. In both cases, de-

velopers perform design using UML notation in a UML compliant tool. This notation is marked up with stereotypes, the UML extension mechanism to type classes, so that developers can relate to a class titled *Agent* with a piece of software that will be generated later. The approach seems promising since it is a quick method for integrating agent techniques with UML, apart of AUML.

Though it is not a development environment, DESIRE (Brazier et al., 1997) deserves a few lines. Probably, it is one of the most mature works in this field, due to the number of related papers and training courses (see <http://www.iids.org>). DESIRE proposes a method to build agents based on the recursive composition of interconnected tasks. DESIRE is supported by a theory on the operation of the framework, a method of development, and tools to facilitate a development with this framework (Brazier et al., 2002). There are examples of the application of DESIRE to different domains, like a scheduler for appointments in a call center (Brazier et al., 1999) or the diagnosis of failures in a fridge (Brazier et al., 2002). Both papers show in detail how a design takes place in this approach.

3.2 Design without a Development Environment

Not using a development environment means that developers have to work more choosing proper theories, methods, and software. To help developers in selecting and applying them into their systems, it would be a good idea to review works that have made this effort, already. For instance, (Massonet et al., 2002) considers how to translate MESSAGE/UML diagrams to a concrete target agent platform, JADE (see chapter 13), where control of agents are expert system shells. For more detailed examples, readers may consult (Bernon et al., 2002; Caire et al., 2001b; Kinny et al., 1996) which correspond to different approaches to this problem: object-oriented, agent-oriented, and formal methods based. These are works that provide examples of use and that integrate results in existing research in agents. There are others that deserve being mentioned, but the lack of space prevents a serious review. Readers are invited to review chapters in this book to find additional information.

Reading these works, developers will find out that there are too many concerns to take into account. Among others, authors of this chapter highlight the selection of an agent model, agent architectures for models of agent, code distribution issues, agent features design, agent platforms, and MAS frameworks.

Though relevant, the problem of selecting an agent model is not addressed here. As it was said in the analysis section, interested readers are invited to consult (Huhns and Singh, 1998) to get a general picture of what features provide different representations. To complement this view, (Nwana, 1996) contains a huge collection of references to developments of agents in many domains. De-

velopers can obtain examples of to know how to develop agents for different purposes.

To introduce other aspects, this paper provides three sections. The first deals with agent architectures. The second with issues related with the distribution of agents, MAS or internal components among different nodes in a network. The third presents research works that addresses theory and implementation of different agent features. The last one collects recommendations of agent platforms and frameworks for MAS design.

Agent Architectures. Why an agent architecture? Because an architecture shows how to put together different pieces of software and make them interact. Here, an agent architecture would provide a framework for realizing different features that researchers require from agents. In this domain, agent architectures have been defined in many ways. Specially, when research on agent started, agent architectures, like IRMA (Bratman et al., 1988), were defined using flows of data among interconnected boxes whose functionality was explained in natural language. There are logical definitions of agents, like those expressed as tuples of functions, as in (Wooldridge, 1992). Each function defines a particular aspect of the agent, like belief revision functions that determine in each state what beliefs are correct. Layered definitions are also frequent, like (Kendall and Malkoun, 1996). In a layer definition, sensory inputs come through lower layers and induce reactions which propagate upwards. When an upper layer wants to perform an action in the environment, the process is the opposite: downwards propagation. Each layer inputs the output of lower layers, and viceversa. Finally, there are also definitions biased by object-oriented approaches that define systems, subsystems, their interfaces and how they are interconnected, like agents described in (Garijo et al., 1998). This kind of definition is very useful towards implementation, since it already identifies the interfaces of components, the number of existing components, and their purpose. In this direction, software engineers also use Architecture Definition Languages (ADL). Reviews on ADL can be found in (Clements, 1996) and (Medvidovic and Taylor, 1997). An ADL expresses at a high level what subsystems exist and how they connect. Let us notice that UML v. 2.0 is being considered as a kind of ADL. This would be very interesting, since it would facilitate the integration of an *architectural view* in the agent approaches close to UML.

In an effort of classifying existing agent architectures, (Wooldridge and Jennings, 1995b) suggested three paradigms: deliberative architectures, reactive architectures, and hybrid architectures. *Deliberative* stands for *thinking before doing*. These architectures usually have a symbolic representation of knowledge and provide mechanisms to decide actions upon it. Though flexible, these architectures have an important drawback: reasoning mechanism consume too

much time. Perhaps when the agent decides what to do, it may be too late. *Reactive* means that there is no reasoning on deciding what to do next, just associations of inputs and outputs, like *should happen A, then do B*. These architectures decide what to do next very fast, but chosen action may not be the best one. Finally, *hybrid architectures* are architectures that share deliberative and reactive features. (Wooldridge and Jennings, 1995b) contains representative examples of architectures of each kind that need not to be mentioned again. Just to add a couple of references, this section mentions some recommendable examples of deliberative, SOAR (see <http://www.eecs.umich.edu/~soar/docs.html>) and Cougaar (see <http://www.cougaar.org>), and a hybrid architecture, INTERRAP (Muller, 1996).

SOAR derived from the original work of A. Newell (Laird et al., 1987) is an important deliberative architecture. There have been applications of SOAR ranging from modelling human behavior in urban combat till players in first-person-shoot-em-up games. Cougaar, according to their experiments, is may be the most suitable agent architecture available nowadays. There are development manuals and examples of developments available at its Web site.

INTERRAP (Muller, 1996) shows in detail how this kind of agent is built and used to solve concrete problems. Rather than software, INTERRAP provides the experience of the developer in how a determined organization of components may increase software reuse from one domain problem to another.

The list of agent architectures may continue for pages and pages. Of course, it is recommendable knowing at least representative examples. To save some extra reading, (Muller, 2003) gives rules of thumb to help selecting a suitable agent architectures. These rules are obtained after a study of several application domains and their key features.

Distribution of Components. Distribution of agents across a network, parts of a MAS, or just internal components of an agent is a decision that may take place at this stage. As an example of how to deal with distribution of agents, readers can consult (Gervais and Muscutariu, 2001). As an alternative to distribution, agents can use mobility and forget about where they are. However, an agent can move from one node to a destination node in a network if and only if there is an agent platform, or similar, installed in the destination node. So, who decides which nodes implement what agent platform so that it all works? Deployment is an important part of a specification. The problem is not new at all. In fact, there is notation in UML to deal with this problem and show how the final system should be deployed.

In any case, distribution of agents among different machines, physical or virtual, implies taking into account the issues about how communication may take place, what is being communicated, and how this communication can be used to organize a system behavior.

- *Communication technologies.* Representative technologies that facilitate communication between components are: shared spaces of tuples, *Remote Procedure Call* (RPC), and message passing. The first is a repository of information where several processes are connected and read/write information. An extended, and free, implementation of this technology is Java Network Interfaces (JNI) from Java. The second is based on the existence of a middleware acting as intermediary among two processes. In this variant, a process implements an API, which is offered remotely to other processes in any machine through this middleware. The main reference in this technology is CORBA (OMG, 2000a), a standard ported to most programming languages and Operative Systems. CORBA provides many more things than RPC, but this is not the place for such a discussion. Another middleware offering similar features as CORBA is .NET (see <http://www.microsoft.com>) which is recently born, compared to CORBA, and based on Microsoft platforms. Also, the famous Remote Method Invocation (RMI) available in Java, that supports serialization and transmission of objects through the network. Finally, message passing, though not new at all, perhaps the most frequently used in agent research. It may relay on any of the previous communications technologies since what it defines is asynchronous high level communication among two processes. Messages themselves are described with *Agent Communication Languages* (ACL).
- *Agent Communication Languages.* An ACL describes the format and semantics of messages interchanged among two or more agents. Interpreting properly an ACL requires more than simply parsing the message and extracting data, as remarks (Genesereth and Ketchpel, 1997), since messages have an implicit semantics that detail what kind of reaction is expected in the receiver. There are nowadays two main ACL: KQML (Labrou and Finin, 1997) <http://www.cs.umbc.edu/kqml> and FIPA-ACL <http://www.fipa.org>. The first established the bases of current FIPA-ACL, identifying a set of speech acts to be used in agent communication. However, current research focuses in FIPA-ACL, that synthesizes the best of KQML together with other aspects of agent communication, such as standard protocol definition, content languages, and ontologies. Readers interested in the semantics of FIPA-ACL are invited to review its specification, where semantics are expressed using modal logics. Both KQML and FIPA-ACL have several implementations accessible from <http://www.cs.umbc.edu/kqml> and <http://www.fipa.org/resources/livesystems.html>.
- *Ontologies.* An ontology determines allowed terms in the content of a message, as well as concrete semantics and relationships with other

elements of the ontology. Specialized languages to define ontologies are Resource Description Framework RDF (see <http://www.w3.org/RDF>), a classic one, and DAML (see <http://www.daml.org/language>), the unofficial successor of RDF nowadays. To handle ontologies, there are tools such as those available at (see <http://www.daml.org/language>) and Protégé (see <http://protege.stanford.edu>). The Web site of Protégé contains libraries of ontologies as well as tutorials, papers, and examples. Also, there are extensions to the JADE agent platform to use ontologies created with Protégé.

- *Coordination.* Coordination languages provide description of how interaction should perform over the time. The importance of coordination in MAS does not need justification (see chapter 14). According to (Gelernter and Carriero, 1992), a system can be built out of a computational model and a coordination model. The first deals with sequences of instructions to be executed without interruption. The second attends to how these pieces appear together so that the system satisfies its initial requirements. To achieve such decoupling of aspects, computation vs. coordination, and their later integration, coordination languages propose both a language to express the coordination and frameworks to support these languages. For a general discussion about relationships between computation and coordination, pros and cons of different integration solutions, readers can consult (Gelernter and Carriero, 1992). In this domain, a main reference is the Linda language (Carriero and Gelernter, 1989). It defines how coordination can be defined when the communication is performed over a shared tuple space. For a complete review of relevant coordination languages, it is recommended to read (Papadopoulos and Arbab, 1998). But, how does a developer apply these languages in a MAS? What is different between conventional coordination languages and MAS is the computational model used by agents, which, in general, is more complex. To help in the adaptation of these concepts, readers are invited to consult chapter 14.

For instance, JADE uses, by default, RMI as internal communication facilities, defines wrappers for FIPA-protocols and other facilities to decouple computation from interaction, and includes components to define ontologies. In this sense, JADE is very complete. However, a designer must know that it is possible to generate other kinds of agent-based systems by reusing existing technology.

Agent Features. Researchers associate agents with certain features like autonomy, social ability, or intelligence. This section gathers references to research work in designing these features. Why considering these elements

in the design section? Because agent abilities are related with agent models and, as it was mentioned before, agent model selection had been postponed to design.

Autonomy, as intelligence, is a term that it is hard to define since it involves philosophical considerations. According to dictionaries, it can be understood as *freedom of will*. However, (Hexmoor et al., 2003), a kind of survey on autonomy, shows that there are many forms of autonomy and different ways of understanding it. Among others, it refers to *adjustable autonomy*, the user of an agent decides whether it is autonomous or not, and *behavioral autonomy* as the agent's capacity to be original and not guided by outside source. But, how to achieve or describe autonomy?

- (Hexmoor, 2001) defines a predicate calculus account of autonomy using a BDI model. This representation clarifies the notion of autonomy through the concept *situated autonomy*:

Situated autonomy is an agent's stance, as well as the cognitive function of forming the stance, toward assignment of the performer of a goal at a particular moment when facing a particular situation.

- (Luck and d'Inverno, 1995) describes autonomy using Z (Spivey, 1992) notation. This description assumes that autonomy arises when an agent has a set of *motivations*. Here a *motivation* is:

... any desire or preference that can lead to the generation and adoption of goals and which affects the outcome of the reasoning or behavioral task intended to satisfy these goals.

- (Castelfranchi and Falcone, 1998) proposes to view autonomy through a theory of delegation. In this paper, an agent is autonomous with respect a task delegated by other agent. Say an agent A has to execute a task demanded by an agent B. Factors to be considered in order to qualify the degree of autonomy of A with respect B according to this delegated task are: how unspecified the task to execute is, who is responsible of checking the state of the task, and what decisions have been delegated.
- (Weiss et al., 2003) introduces a formalism called *Role-Norms-Sanctions* (RNS) for explicit specifying the autonomy of computational agents. The idea behind RNS is to support developers of agent-oriented systems in precisely stating what an agent as an autonomous entity is allowed, obliged and forbidden to do. RNS is supported with a tool called XRNS, so developers can generate rather easily their definitions.

Artificial Intelligence and Distributed Artificial Intelligence are the main disciplines in studying computational aspects of intelligence. (Russell and Norvig, 1995) is a rather complete review of modern AI research in this topic. It focus on AI applied to the construction of intelligent agents. With respect

DAI, (Ferber, 1999) makes an account of relevant research of DAI in the context of MAS. Also, (Weiss, 1999) contains a broad review of research in DAI and MAS considering key topics such as distributed problem solving, reasoning, or MAS learning.

As a guideline for reviewing computational intelligence, a broad field, this section follows list of relevant topics in intelligent agents according to (Russell and Norvig, 1995). These are planning, Problem Solving Methods, Learning and Reasoning. For a deeper understanding of the reviewed topics, (Ferber, 1999; Rich and Knight, 1990; Russell and Norvig, 1995; Weiss, 1999) should be consulted. Developers looking for software for any of these topics, can go to <http://www.cs.cmu.edu> and download it. This address also contains links to relevant material in AI.

- *Planning*. Incorporating planning capabilities means that an agent will know, without human intervention, how to combine different tasks in order to get a concrete result. There is a lot of research in planning in AI. STRIPS (Fikes and Nilsson, 1971) is one of the seminal works in the area. It provides a planning algorithm and a language to describe tasks. In the agent domain, TAEMS (Decker, 1996b) and Generalized Partial Global Planning (GPGP) algorithm (Decker, 1995) must be mentioned. The first provides concepts and notation for the second, which is a distributed planning solution for tasks that assumes that participants in a plan may have only partial information about it. For both works, there are software and examples available at <http://dis.cs.umass.edu>. A more comprehensive list of planning systems and architectures can be found in (Amant, 2003).
- *Problem Solving Methods (PSM)*. A PSM describes the reasoning of knowledge-based systems as patterns of behavior that can be reused across applications (Fensel and Motta, 2001). Using PSM can make a program autonomous since it provides reusable behaviors to solve concrete problems. There are libraries of PSM and methodologies that integrate them into a development lifecycle. For a general overview of what a PSM is, from a Knowledge Engineering point of view, and what kinds are available, readers can consult (Fensel and Motta, 2001). For an adaptation of PSM to the agent domain, readers should consult MAS-CommonKADS (Iglesias et al., 1998a), specially the section dedicated to *task model* and *knowledge modelling* which addresses PSM integration. For an extended version, readers can consult original work in (Iglesias, 1998). Finally, extended surveys on distributed PSM can be found in (Decker et al., 1989; Durfee et al., 1989).
- *Learning*. Learning is a key element if the developer wants the agents to improve over the time. There are many kinds of learning techniques.

Apart from AI literature, readers can consult the machine learning review (Dietterich, 1998). That paper contains pseudo-code examples for different kinds of algorithms and studies of their performance and results. In the agent domain, readers can find the chapter (Sen and Weiss, 1999), that contains learning techniques applied to MAS, and (Stone and Veloso, 2000), that offers a collection of different MAS scenarios and a collection of useful learning techniques to apply in each of them.

- *Reasoning.* In developing reasoning mechanisms, logics have proven to be a valuable tool. Nevertheless, reasoning capabilities depend strongly on the kind of knowledge representation. For instance, with a first-order logic representation, there exists algorithms allow to draw conclusions or courses of action. It is not exactly planning, since it deals more with theorem proving techniques and logic inference (backward or forward chaining). To know more about trends in reasoning using logics, readers can consult (Halpern et al., 1995). For a comprehensive review of knowledge modelling techniques, readers can consult (Devedzic, 1999). That paper collects examples for each kind of representation as well as references to tools that help in modelling and reusing knowledge.

An agent being social, according to (Wooldridge and Jennings, 1995a), meant that an agent had to interact with other agents using an Agent Communication Language (Genesereth and Ketchpel, 1997). Results referred here are closer to (Huhns and Stephens, 1999) point of view. In (Huhns and Stephens, 1999), being social implies characterizing agents with abstractions from sociology and organizational theory. In principle, this kind of view would facilitate developing MAS using peer-to-peer interaction instead of a server-client paradigm. This section will comment results in two research lines on social aspects: how to make an agent conscious of its relationships with other agents and how to build a society of agents. The first line refers to representation of social dependencies and reasoning upon this information. The second line distinguishes between agent being organized and agents organizing by themselves, also known as *self-organizing* (see chapter 17). For an alternative point of view about these aspects, readers are invited to consult (Boissier, 2003). (Boissier, 2003) presents organizations in detail from an observer point of view, from the designer point of view, and from the perspective of an agent.

(Sichman et al., 1994) introduces the study of social dependencies. This paper indicates, starting from an external description of an agent, how to derive dependencies upon other agents and establishes the foundations of the so-called *social reasoning*. This kind of reasoning is presented in more detail in (Sichman and Demazeau, 2001) as:

... a mechanism that uses information about the others in order to infer some new beliefs from the current ones.

This research links with other aspects already reviewed in this paper, such as autonomy or coordination. According to authors, these results had been implemented in two systems: DEPINT (Sichman, 1998), a simulator of micro-societies, and DEPNET (Conte and Sichman, 1995), an open MAS.

In the *agents being organized* trend, the first recommended paper is (Ferber and Gutknecht, 1998). In this trend there is an organization that can be distinguished as an first class citizen or just appear in form of roles and social dependencies. (Ferber and Gutknecht, 1998) presented a meta-model for organization description where organization appeared as first class citizen. This work evolved into the MADKIT platform, that will be reviewed later. A similar organization description appears in MESSAGE/UML (Caire et al., 2001b), where organizational entities relate with interaction related or task related entities. Gaia (see chapter 4) describe its organization with three main organizational abstractions: organizational rules to constraint system behavior, organization structures by defining roles, and organizational relationships that determine agent to agent dependence. This notion of the organization is partially shared by *enterprise modelling* (Fox and Gruninger, 1998). This discipline studies how to create computational representations of businesses, governments, or any other organizational structure. In this discipline, readers can find ontologies to describe MAS organizations as well as tools to simulate organizations or formats to interchange organizational processes definitions. The application of agents in enterprise modelling to simulate or implement business process workflows have appeared with relative frequency in the agent literature. (Stohr and Zhao, 2001) introduces basic workflow terminology, concepts, and related architectures. It is a good first step towards understanding what is the role of agents in workflows. Examples of application of agents to workflow automation are (Walshe et al., 2000), that shows how to define a workflow using XML and reactive/cognitive agents, and (Judge et al., 1998), that shows how agents can enhance a workflow basic functionality.

In the *agents organize by themselves* trend, the main concept we have to deal with is *self-organization*. Self-Organization approaches assume that agents get organized without human intervention. This idea is the opposite of the previous one, where organization exists by itself, whereas here it is just a bottom-up construct. This had been discussed deeply in philosophy in relation with the concept of *autopoiesis* and biological systems, readers can check (Whitaker, 2003) (see <http://www.calresco.org/sos>) for an introduction and related software. Here, researchers look for *emergent behaviors*, or how to obtain certain patterns of global behavior in a MAS by changing the way some agents interact. As an example, readers can review (Roli, 2002), a work based in cellular automata. A cellular automata are cells in a grid that can only switch on or switch off autonomously depending on the state of their neighbors. More examples of self-organization and emergent behaviors can be found in literature

about *MultiAgent-Based Simulation* (MABS) like (Conte et al., 1998; Moss, 2000), both gather research trends in MABS. There are some attempts of structuring these *self-organization results* into bodies of knowledge. One of them is ADELFE (Bernon et al., 2002), a methodology supported by the ADELFE toolkit. This methodology, with a dedicated chapter in this book, integrates *Adaptative Multiagent Systems* (AMAS) considerations in the development process so that developers can have some control over this kind of systems. Also, there is *Engineering Self-Organizing Applications* (see <http://gaper.swi.psy.uva.nl/esoa/content/main.php>) working group. This group, as its name remarks, researches self-organization applied to agents. At their Web site, researchers will find references to work on this kind of systems and an interesting survey of their current results.

A work in the middle of the previous trends is (Esteva et al., 2002). This work provides methods to obtain emergent behaviors that satisfy the needs foreseen by developers at design time. Readers interested in this work can refer to chapter 10 dedicated to *Social Agents Design Driven by Equations* (SADDE) methodology.

Besides organizational approaches, there are other alternatives to manage the behavior of a MAS, like policies. In networks and telecommunication domain, the concept of *policy* is widespread. A *policy* describes a constraint in the behavior of the system. This concept of *policy* would be similar to the *social laws* (Shoham and Tennenholtz, 1995). The main difference is that a *policy* can be changed in runtime. The application of *policy* to agents has been studied in (Dulay et al., 2001). This paper shows a language to define policies and a framework based on agents to support them. The work is interesting because it shows researchers how to build a system that support behavior changes in runtime.

Agent Platforms and Frameworks. Citing (Pree, 1995), frameworks represent application skeletons for a particular domain. The utility of a framework, then, it is to save the effort in developing by reusing architectures, components or libraries. An *agent platform* can be understood as a set of services that allow agent management and communication. An agent platform may come with shells of agents that developers can reuse for their systems. There are so many that reviewing all existing MAS frameworks in this chapter is not possible. Thus only some of them will be mentioned here. For a broader view, readers are invited to go through other surveys like the list of agent software from <http://www.agentbuilder.com> or *software reports* and *technology roadmaps* available at AgentLink (<http://www.agentlink.org>). As developers, adopting tools that follow standards is a wise option. However, in the agent domain, standards do not provide answers to some problems, like

what the internal structure of agents is or its control. That is the reason to consider not only standards, but also other proposals:

- **Standard.** There are two standards MASIF and FIPA. MASIF stands for Mobile Agent System Interoperability Facility (MASIF) (OMG, 1999) standard. It is based on pure CORBA to implement communication among agents and mobility. The official platform for MASIF is Grasshopper (see <http://www.grasshopper.de/index.html>). FIPA Stands for Foundations for Intelligent Physical Agents. It proposes a number of services similar to CORBA, but centered on defining interaction protocols and communication languages. Also, FIPA services can be implemented over different communication technologies, like RMI, CORBA, or HTTP based. JADE (Bellifemine et al., 2001) (see <http://jade.cselt.it>) and FIPA-OS (see <http://fipa-os.sourceforge.net>) implement FIPA standard. Other implementations of FIPA are available at <http://www.fipa.org/resources/livesystems.html>.
- **Non-standard.** A frequently referred work is RETSINA (see <http://www.cs.cmu.edu/~softagents>). This framework deals with advanced matchmarking capabilities (Sycara et al., 1999) that facilitate locating agents relevant for a certain task. RETSINA also indicates a kind of MAS infrastructure by identifying general tasks to be accomplished in the system, and roles that tend to appear, like information brokers. Readers will find that RETSINA has been applied in several projects and lots of papers are available about different aspects of RETSINA. Also, RETSINA supplies software to create agents in their Web site.

MADKIT (<http://www.madkit.org>) is an extensible framework for design, in which the graphical front-end is just an optional plugin; it is based on the AAladin framework already mentioned in the analysis section. It provides a basic kernel of MAS where concrete functionality can be plugged in. Also it includes graphical tools to launch, monitor, or kill agents.

In case readers are interested in developing agents using AI approaches, a good choice is ABLE (see <http://www.alphaworks.ibm.com>). It can define agents in terms of predefined modules that implement several control mechanisms ranging from neural networks to decision trees. It also implements learning algorithms, like inference learning. These components can be interconnected in a graphical front-end or manually by a couple of lines of code. This framework is a good tool if the developer wants to experiment how to control an agent by combining different techniques.

The number of non-standard platforms is appealing. Previous ones have been selected due to their impact and success in dealing with different agent features. In any case, the platforms referred here are only a small percent. Readers are strongly advised to consult reviews of agent platforms such as (Ricordel and Demazeau, 2000), that centers on the suitability of agent development tools from a methodological point of view, or (Mangina, 2002), that is an exhaustive survey of MAS development frameworks.

4. Implementation

Implementation is the translation of design concepts to programs compilable to executable code or interpretable. To implement a MAS, the language may be conventional or agent-oriented. This chapter assumes that, despite the high level of abstraction of agent oriented languages, they are still programming languages. Therefore, proposing a development using exclusively an agent oriented language, without specialized analysis notations, or without taking into account design concerns, may be affordable in a small development. But, the same approach in a medium size may not be realistic.

This section does not address the problems of reusing software (agent platforms, MAS frameworks, development environments). It is assumed that these elements have already been selected in the design and their influence taken into account. So the problem is to select a language to implement these components. If the developer is using some existing software, perhaps there is no choice.

- Declarative languages (functional and logic-based).
 - April (McCabe and Clark, 1995) is a functional language that incorporates communication facilities. Examples of April can be obtained from <http://www.nar.fujitsulabs.com>. Reference manuals available from April Web site contain examples of application.
 - Concurrent-METATEM (Fisher, 1995) allows to express temporal logic programs. A previous paper (Fisher, 1994) shows applications of this language and some small examples.
 - CLIPS stands for C Language Integrated Production System. It is an expert system shell that can be used to implement the behavior of an agent. The behavior can be expressed using rules, as it is done in some works referred in this chapter. Tools and manuals for CLIPS can be downloaded from <http://www.ghgcorp.com/clips/CLIPS.html>. JESS (Friedman-Hill, 2003) is a port to Java of CLIPS. JESS has been used into JADE agents as an alternative to the default JADE behavior definition.

- Mozart (see <http://www.mozart-oz.org>) is a (multiparadigm) language. In concrete, it is a functional concurrent object-oriented language with some enhancements to support mobility, (van Roy and Haridi, 1999) is a review of some agent-based projects using Mozart.
 - Prolog is a pioneer language in logic programming. Its origin is quite complex and many researchers have worked on it. (Cohen, 1988) contains a review of relevant research work involved in the creation of this language. There are many implementations of this language and suitable extensions. (Sadri and Toni, 1999) contains references to extensions of this language to define agents. Each extension provides its own examples of application.
 - Lisp (McCarthy, 1978) is one of the first languages in AI. There is a huge collection of libraries and utilities based on Lisp (<http://www.alu.org>). This site also contains references to applications of Lisp to agents, like LISA (see <http://lisa.sourceforge.net>), a production-rule system implemented in Lisp, or OSCAR (see <http://oscarhome.soc-sci.arizona.edu>), an architecture for anthropomorphic agents.
- Agent-oriented languages. They incorporate concepts common to agent theories but do not provide primitives dealing with concurrence or temporal logic.
 - ConGOLOG (De Giacomo et al., 2000) is a language based on situation calculus. It is the concurrent version of GOLOG. It is downloadable from <http://www.cs.toronto.edu/cogrobo/systems.html>, where development examples are also available. For more details on ConGOLOG, readers can consult (Lesperance et al., 1999) where authors model a mail-order business.
 - Agent0 (Shoham, 1993) is the first agent-oriented language. Unfortunately, there are no interpreters available. PLACA (Thomas, 1995) is frequently referred as an extension of Agent0 to MAS. It incorporates ideas of planning among several agents. As Agent0, it cannot be found in the Internet.
 - AgentSpeak(L) (Rao, 1996) is designed as a definition language for BDI agents. SIM SPEAK (Machado, 2003; Machado and Bordini, 2001) is an implementation of AgentSpeak(L). There is an extension named AgentSpeak(XL) (Bordini et al., 2002) that incorporates elements from TAEMS (Decker, 1996b) to the original work.

- MAML (Gulyas and Corliss, 1999) is a language to model social simulations of MAS. It is derived from SWARM (see <http://www.swarm.org>), another language for MABS. Tutorials, software, and examples can be accessed at <http://www.maml.hu>.
- 3APL (Hindriks et al., 1999) stands for *An Abstract Agent Programming Language*. This language has been defined to express control and deliberation in BDI agents. There is software, tutorials, and examples available at 3APL Web site <http://www.cs.uu.nl/3apl>.

Of course, a developer can choose any conventional language: structured or object-oriented. In these cases, the developer has to rely on libraries that provide basic functionality, like communication facilities. Most of the agent platforms presented in previous sections are implemented using object-oriented paradigm. However, it may be needed to integrate different programming paradigms into a single architecture. In that case, there are three possibilities: wrapping the foreign language, creating mediators among these structures implemented in different languages, or simply rewriting foreign code. Wrapping is the solution to integrate expert system shells and agent architectures, as the JESS and JADE integration. For mediators, a low cost solution is to use middleware, like CORBA. Mediators of components implemented in different languages offer a remote interface, which is accessible using the same facilities. Rewriting should be the last option, since it requires a lot of work.

In combination with these languages, the developer can use other purpose specific languages. There are languages specially designed to cover issues like coordination, knowledge representation, or ontology representation. References to such languages have been already mentioned in the previous section dealing with architectural issues.

5. Testing

Testing enables to identify the existing failures and to check if the code sticks to the specification of the system or at least, if it satisfies the requirements of customers. In this stage, classic software engineering distinguishes between validation and verification. Citing (Boehm, 1984), in the case of validation, an engineer focuses on the question “*Am I building the right software?*,” whereas in the case of verification, the focus is on the question “*Am I building the software right?*” In other words, verification is concerned with the (formally) checking the internal consistency of specifications, and validation is concerned with checking the specifications’ consistency with the stakeholder’s intentions.

Research in testing MAS has mainly addressed verification aspects. Validation of MAS has been studied in works related with agent-oriented require-

ments engineering, like i^* and Tropos. As it was reviewed in the analysis, software requirements are labelled as *goals*. As they appear as first-class concepts of the specification, it is easy to say if some requirement has been considered or not. It is a matter of checking whether a goal has any associated activity or not. This naive approach does not work in all cases. For an example of more complex validation using a AORE approach, readers can consult (Dubois, 1998). This paper contains an example of validating a system using the specification language Albert II. The paper also describes a tool able to simulate the specification so that clients can see how it should work and perform the validation themselves.

Since there is more literature dealing with verification than validation in the agent domain, verification will be considered in detail in the next section. The last section will be dedicated to study debugging approaches with agents. Debugging is the complementary task of validation and verification. It seems coherent to reference also research works and tools that help developers to find out exactly why the program has failed.

5.1 Verification of MAS

As an introduction to conventional software engineering testing techniques, readers can consult (Pressman, 1982). The most relevant ones are *black box* and *white box* tests. The first considers the system as a black box where only its inputs and outputs are known. The second is quite similar, although it assumes a certain knowledge of internals of the black box. In MAS, white and black box testing is rarely published as original research. Though still infrequent, MAS formal verification is more likely to appear.

Formal verification is based on the existence of an specification expressed with a formal language. In the analysis section, there was a review of different languages that could be used for formal specification. The verification itself can be applied anytime. A verification usually demonstrates the correctness of a program with respect to a specification of what it has to do. For instance, based on Petri Nets formalisms, (Xu et al., 2002) determine if a plan performed with the collaboration of several agents can be performed. For this, they input a Petri Nets specification of the plan. On the other hand, to verify communication from AUML diagrams, (Poutakidis et al., 2002) suggest a mapping method from AUML diagrams to Petri Nets, which well-known algorithms of deadlocks detection are associated with.

According to (Wooldridge and Ciancarini, 2000) there is little work in the verification of MAS. Existing works could be categorized into axiomatic approaches and model checking approaches. The first have several variants and all of them can be considered as theorem proving problems. Research in automated theorem proving field started early in the last century. Unfortunately, it

requires high skills in logics for those interested in applying it. For an overview of what kinds of theorem proving techniques exist, readers can consult (Bledsoe, 1985), a survey frequently referred in the area. With respect to application of theorem proving in MAS, (Wooldridge and Ciancarini, 2000) cites works in which axiomatic verification is applied to MAS specified with BDI logics and Concurrent METATEM. However, as (Wooldridge and Ciancarini, 2000) notice, a main problem here is how to apply this kind of verification when the BDI principles are implemented with non-logic based languages, such as C++ or Java. This is known as the *computational grounding* problem.

Let us also mention the trend called *design by contract* (Meyer, 1992) that consists in defining pre/postconditions and invariants for the methods or procedures of the code and verifying them in runtime. Violating any of them raises an exception. This technique is built-in in last versions of Java and can be simulated in C++ (Plosch and Pichler, 1999). The problem is this technique does not check program correctness, it just informs that a contract has been violated.

Model checking is less fine grained than axiomatic approaches, but also more tractable. It consists in verifying concrete properties that a system must satisfy. The method inputs a model of the system to check and a property definition. Of course, the language in which model and property are defined is relevant. This approach seems to be well accepted by industry. Its difficulty is the identification of the interesting properties to check. Some of them are listed hereafter, extracted from the literature:

- *Liveness*. The agent always has something interesting to do. According to Wooldridge's model (Wooldridge, 1992), this is related to the *weakly complete* concept: there should always be at least one applicable action or message available to every agent, whatever its beliefs. This property is relevant because a developer surely wants that the agents do not get stuck.
- *Deadlock free*. A MAS, or some agents belonging to it, may fall in different kinds of deadlocks. A deadlock means, in general, that an agent is blocking others and being blocked at the same time. This may happen, for instance, when there are shared resources among agents and an agent requires a resource that the other holds and viceversa, but none of them wants to release them. There have been thorough deep studies of deadlocks in the concurrent programming, operating systems and telecommunication literature. To have a deeper knowledge on deadlocks, theory and practice, it is very recommended reading the classic (Coffman et al., 1971). This paper focuses on deadlocks mixing both theory and practice on detection and prevention.
- Another interesting property is that a task must not take the system to an undesirable state. This is a property required in the situation calcu-

lus approach of ConGOLOG (De Giacomo et al., 2000). Unlike other properties, ConGOLOG itself takes care of this aspect. For a developer, this property has a high value, since it ensures that state of the world always changes according to the desires of the developer. However, in real systems, this property is very hard to ensure.

Some free software is available to perform model checking and theorem proving. (Bowen, 2003) contains references to this kind of software. As an example of how to apply model checking to MAS, apart of those in (Wooldridge and Ciancarini, 2000), readers can consult agentTool (DeLoach, 2001) and MABLE (Wooldridge et al., 2002a). Both use SPIN model checker (Holzmann, 1991). agentTool uses SPIN allowing to identify deadlocks in agent interaction. MABLE is a programming language enabling to include asserts in the code. These are named *claims*. SPIN is used to verify their truthfulness. It must be said that MABLE is different from the *design by contract* solution, that is verified only in runtime, whereas SPIN verifies during compilation.

Model checking, in principle, requires translating the system into a model specified with a concrete language. SPIN uses as input a language named PROMELA (Holzmann, 1991). However there are works that suggest that this is not a compelling condition. For instance, (Visser et al., 2000) have studied how, from source code in C++, extract a model of the behavior of the agent, by adding some instructions to the original code, and then perform the model checking.

5.2 Debugging MAS

What if something goes wrong? Researchers in debugging distributed systems give some answers. Debugging MAS is similar to debug open distributed systems or concurrent systems. As developers know, an important problem of distributed systems is that there is too much information to analyze. There is literature that deals with this problem, like (Garcia-Molina and Germano, 1984) or (Joyce et al., 1987). (Joyce et al., 1987) describes the construction of a monitoring system and how collected information can be presented to the user. Benefits of visual representations for debugging is also discussed in (Baecker et al., 1997), where authors propose different kinds of visual and audio presentations of source code and insights of a program. In other trend, (Garcia-Molina and Germano, 1984) proposes generation and later analysis of traces of programs.

In the agent domain, there are few tools that can help to test and then debug a system:

- Zeus incorporates visual debuggers to view internal state of agents. Authors of Zeus extend their ideas in (Ndumu et al., 1999) proposing internal and external inspectors for a MAS.

- JADE has a *sniffer agent* that shows in a separate GUI what ACL messages are exchanged by the agents.
- MADKIT (Gutknecht and Ferber, 2001) uses graphical tools and introspection on agent code to discover at runtime groups and roles, references to other agents, and other direct manipulation of these structures.

The drawback of these tools is that they must be used since the beginning of the project. So, what if the framework or the agent platform is none of these? Then, the developer should consider using the techniques commented at the beginning of the section.

6. More Information

To broaden the vision of the field, readers are invited to consult other surveys. As a recommendation, it is suggested (Jennings et al., 1998; Nwana and Ndumu, 1999; Weiss, 2003) (the latter served as a main source of inspiration for this chapter).

Current trends in software agents are well reflected in the AgentLink Web site <http://www.agentlink.org>. Under AgentLink cover, there are regular publications like the *AgentLink Roadmaps*, published each year, and special interest research groups on different topics, like the *Methodologies and Software Engineering for Agent Systems* (MSEAS) group, which is specialized in methodologies. AgentLink also sponsors *European Agent Systems Summer School* (EASSS), a school whose proceedings contain tutorials about relevant agent topics.

The Object Management Group, responsible of UML and CORBA standards, has a special interest group on agents (see <http://www.objs.com/agent/index.html>) with links to projects and documents.

With respect to conferences, let us mention *Autonomous Agents and Multiagent Systems* (AAMAS) conferences, which are of very high quality and the *AOSE workshop*, focused on software engineering for agents. There are also chapters dedicated to MAS research in most conferences on Artificial Intelligence, like ICAI, ECAI, or IBERAMIA, whose proceedings appear in catalogues of relevant publishers.

7. Conclusions

This chapter has introduced briefly research results that can help developers and MAS researchers to create MAS. It has surveyed which tools, software libraries, frameworks, theories, and methods are available today for developers. Indeed, having so many results is good news since a developer can produce a MAS with less effort than years ago. However, there are still important gaps and questions, like how to jump from agent theories to MAS implementation,

which are the consequences of selecting a concrete agent architectures, how to reuse existing MAS development experience in other developments, or what concepts are needed to tackle with each aspect of a MAS. The agent community is making a huge effort to answer these and others questions, and this is not a trivial task at all. So the best way to finish this chapter is simply congratulating researchers for the work done and encouraging them to keep on contributing to this field.

Acknowledgements

Gerhard Weiss greatly acknowledges support by the German National Science Foundation (DFG) under contract Br609/11-2. Jorge J. Gomez Sanz acknowledges support by Spanish Ministry of Science and Technology under grant TIC2002-04516-C03-03 and Ruben Fuentes and Juan Pavon for reading this chapter and providing useful comments.

II

METHODOLOGIES FOR AGENT-BASED SYSTEMS DEVELOPMENT

Introduction

Because MAS are relevant to solve complex and distributed problems, software to be designed is more and more complex. This complexity comes from different difficulties such as:

- Identifying the task the global system has to solve;
- Identifying what entity in the application is an agent;
- Defining interactions between agents;
- Specifying adequate and/or complex protocols;
- Defining interactions between the system and its environment; and
- Defining the agent's relevant behavior.

Software to design deals with new concepts such as: agent, goal, task, services, organization, interactions, environment, etc. New tools and models are therefore needed to help engineers to work with these new notions. Furthermore, software contains a huge amount of code lines and their distribution increases the complexity for designers who are obliged to take into account new problems such as mobility or security. Methodologies must help designers to deal with these problems and have to manage this complexity. Moreover, in order to have, in a near future, agent and MAS developed in the industrial world, designing methodologies to facilitate and to support agent-based systems engineering is becoming more and more important. A methodology should facilitate the software engineering process by providing a rigorous process which enables the generation of a set of models describing, using a precise notation, the different aspects of the software to be designed. An agent- or multiagent-based methodology consists of a process, a modeling language, or notation, and tools to support the process and the notation and to help designers. At present, it is a widely accepted fact that the main phases of the process are

similar to those used in object-oriented methodologies: requirements, analysis (or specification), design, development (or implementation) and deployment of the system. All the phases of the process are using models and are divided into steps. The life cycle of the development process is generally not sequential but rather iterative and regular backtracks are needed. During the different steps of the process, the modeling language enables designers to express all the models and all the elements that are needed to define the system to be done and to realize verifications during the software engineering. Standardizing the notation at this level or using standards such as UML are important. A methodology generally provides tools, that are associated with the notation, in order to support the graphical notation and to realize some consistency validation. Because agent and multiagent oriented methodology is a new field of research and because the new paradigms of agent are associated with new concepts, the main works focus on the first three phases of the process: requirements, analysis and design. But, there are a lot of works related to agent platforms which can be used in the development and deployment phases. This part of the book deals with three well-known and general methodologies in particular:

- Chapter 4, “The Gaia Methodology” by Luca Cernuzzi, Thomas Juan, Leon Sterling and Franco Zambonelli describes the key concepts of the Gaia methodology. The analysis and design phases of software engineering are using five models: role, interaction, agent, services and acquaintance models. Then, the authors are focusing on three extensions to overcome the limitations of the first version of Gaia. The new version enables the development of open systems and uses standard notations such as UML and AUML. The environmental model is defined in the analysis phase and the organizational rules and the organizational structures are taken into account in the design phase.
- Chapter 5, “The Tropos Methodology” by Paolo Giorgini, Manuel Kolp, John Mylopoulos and Marco Pistore gives an overview of Tropos. It describes the main phases of software engineering such as: the requirement analysis, the architectural design and the detailed design. Tropos is characterized by the fact that it focuses on the requirements phase and that all the development process is based on mentalistic concepts of agents such as agent, role, goal, task, etc. Several tools help designers in the different phases of the development. A specification language, Formal Tropos, associated with a tool, T-Tool, enables the verification of errors, ambiguities and under-specification during the requirements analysis.
- Chapter 6, “The MaSE Methodology” by Scott Deloach, presents this methodology based on UML notation. The author focuses on the analysis and the design phases. During analysis, concepts like goals and roles

are used. The design phase enables the definition of the agent classes, the agent architecture and the coordination protocols between agents. This phase ends with a deployment diagram. The methodology is supported by the agentTool software which provides some verification of the protocols and enables automatic code generation.

The last chapter of this part presents multiple criteria to evaluate and compare existing methodologies and in particular the three described in this book. Chapter 7, “A Comparative Evaluation of Agent-Oriented Methodologies” by Arnon Sturm and Onn Shehory proposes a framework composed of multiple dimensions and a metric to evaluate and compare agent-based methodologies. The different axes proposed to evaluate a methodology are: concepts and properties related to agents and MAS, notation and modeling techniques, development process and pragmatics. Gaia, Tropos and MaSE are evaluated with this framework. In a general way, the obtained results are quite positive.

Chapter 4

THE GAIA METHODOLOGY

Basic Concepts and Extensions

Luca Cernuzzi, Thomas Juan, Leon Sterling and Franco Zambonelli

Abstract Gaia (Wooldridge et al., 2000b) was the first complete methodology proposed for the analysis and design of MAS. However, the original version of Gaia suffered from the limitations of being suitable for the analysis and design of closed MAS and of adopting non-standard notation techniques. Several extensions to the basic Gaia methodology have been recently proposed to overcome these limitations. In this chapter, we summarize the key characteristics of the original Gaia methodology and present three extensions that have been proposed to improve Gaia and make it more suitable for the development of open MAS in complex environments.

1. Introduction

Gaia was the first complete methodology proposed to guide the process of developing a MAS from analysis to design.

The first version of Gaia, described in (Wooldridge et al., 2000b), emphasizes the necessity to identify proper agent-oriented abstractions around which to base the process of MAS development. Gaia outlines the suitability of the organizational metaphor. A MAS is conceived as a computational organization of agents, each playing specific roles in the organization, and *cooperating* with each other towards the achievement of a common application (i.e., organizational) goal.

The Gaia methodology has been quite influential over the past few years. However it suffers from several limitations that may undermine the possibility of its effective adoption for the majority of real-world multiagent scenarios.

A first limitation derives from the fact that Gaia, in the original proposal, is suitable only for the analysis and design of *closed MAS*, in which agents must be benevolent to each other and willing to cooperate. Unfortunately, this is not

the case for many MAS, where agents can belong to different stakeholders and can express self-interest in actions.

A second limitation relates to the fact that the *notations* used by Gaia to model and represent a MAS and its components appears unsuitable to tackle the complexities of real-world systems and, even worse, do not follow accepted software engineering standards.

As a consequence of the above limitations, *improvements* to the basic Gaia methodology have been proposed in order to both capture the characteristics of open MAS in complex open environments and to improve its notation techniques. This chapter briefly presents the original version of the Gaia methodology and three proposals for extensions: two proposals extend the basic process of Gaia to make it suitable for open MAS, while a third proposal integrates standard notation techniques in Gaia.

To exemplify the concepts expressed in this chapter, we exploit a simple running example in the area of *agent-mediated marketplaces*, as a typical example of an open MAS in which agents may exhibit self-interested behaviors. In agent-mediated marketplaces, agents interested in buying and selling specific classes of goods will meet together to access an environment made up of “wanted requests” and “sales offers.” Transactions typically take place in the form of open public auctions. Agents meeting at a marketplace will form dynamic and open organizations in which agents themselves can play roles such as “client” and “provider” when publishing requests and offers for goods, as well as roles such as “bidder” and “supplier” when subsequently involved in an auction. The intrinsic openness of the marketplace, where different agents each with its own goals may enter to negotiate, raises issues of controlling proper ways to conduct negotiations, i.e., avoiding agents cheating with each other. Moreover, it is possible to conceive of several interacting organizations co-existing in a marketplace. For example, one could have two independent organizations for dealing with the auction phase and the subsequent payment and delivery phases.

2. Gaia in a Nutshell

The first version of the Gaia methodology is described in (Wooldridge et al., 2000b). The scope of the methodology includes the analysis and design phases and exclude both collection of specifications and implementation. It is applied after the requirements are gathered and specified.

In general, the Gaia process consists in constructing a series of models, as shown in Figure 4.1. The models are aimed at describing both the macro (societal) aspects and the micro (intra-agent) aspects of a MAS, generally conceived as an organized society of individuals (i.e., a computational organization of autonomous entities). In the analysis phase, the role model and the interaction

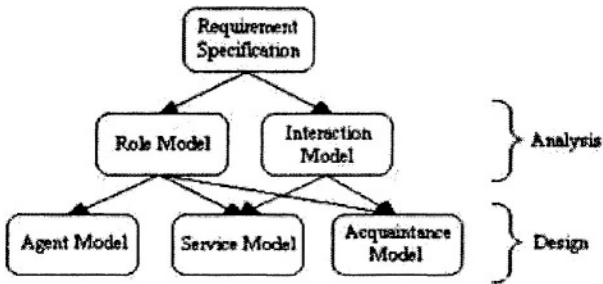


Figure 4.1. Models in the Gaia methodology (Wooldridge et al., 2000b)

model are constructed, depicting the system as a set of interacting abstract roles. These two models are then used as input to the design stage, in which an agent model, a services model, and an acquaintance model are defined to form a complete design specification of the MAS to be used for the subsequent implementation phase (not dealt with by Gaia).

2.1 Analysis with Gaia

In the analysis stage, roles in the system are identified and their interactions are modeled.

Roles are abstract constructs used to conceptualize the system, with no concrete counterpart in the implemented system. In Gaia, all roles are atomic constructs and cannot be defined in terms of other roles. A role schema is intended to be a semi-formal description of an agent’s behavior, and the collection of role schemas for a system define the complete role model. For each role, a role schema is defined in terms of four attributes: permissions, responsibilities, activities and protocols.

Permissions express the environmental resources available to the role, usually in terms of the information that the role can read, write or create. The permissions specify both what the role can and cannot use.

Responsibilities of a role define the role’s actual functionality. There are two types of responsibilities, safety properties and liveness properties. Safety properties are properties that the agent acting in the role must always preserve. These are expressed as predicates over the variables/resources in the permissions of the role, specifying the legal values these variables/resources can take. Liveness properties describe the “lifecycle” or generalized behavior pattern of the role. Liveness properties are represented by a regular expression (see

Table 4.1 for the syntax of liveness properties) over the sets of activities and protocols the role executes. There, activities are intended to represent those tasks or actions a role can take without interacting with other roles, while

Table 4.1. Partial syntax of liveness properties

Operator	Interpretation
$x.y$	x followed by y
$x y$	x or y occurs
x^ω	x occurs indefinitely often
$[x]$	x is optional
$x y$	x and y interleaved

Role Schema: BIDDER		
Description: Evaluating sellers' offers and making a price offer for a service and/or goods in the Auction model.		
Protocols and Activities: <u>ReceiveCfO</u> , <u>Not-Understood</u> , <u>PriceOffer</u>		
Permissions:		
reads	<i>offers</i>	// offers from the seller
changes	<i>offer_evaluation price</i>	// evaluation of the sellers' offers // price proposed
Responsibilities		
Liveness:		
	BIDDER = (ReceiveCfO.PriceOffer)^w	
Safety:		

Figure 4.2. Schema for role BIDDER

protocols are tasks or actions a role can take that involve interaction with other roles.

For the purpose of our running example of the agent marketplace, we identify five possible roles: Client, Bidder, Provider, Supplier, and Auctioneer. In an actual implementation of the system, it is expected that the roles Client and Bidder will be played by buyer agents, Provider and Supplier by seller agents, and that the role Auctioneer will be played by auctioneer agents. However, the Gaia analysis phase abstracts from the presence of agents playing specific roles, an issue that is dealt with in the design phase. Three of those schemas are presented according to the Gaia notation.

Taking a closer look, in Figure 4.4 an Auctioneer needs to access the offer presented by a seller and to propose to the seller the highest price offered by bidders, as stated in its permissions. The liveness expression, that may occur 0 or more times, specifies that whenever an agent implementing the Auction-

Role Schema: SUPPLIER	
Description: Proposing a service and/or goods in the Auction model.	
Protocols and Activities: ServiceProposal, ReceiveApproval	
Permissions: changes offer_definition // service and/or good proposed	
Responsibilities	
Liveness: SUPPLIER = (ServiceProposal)	
Safety:	

Figure 4.3. Schema for role SUPPLIER

Role Schema: AUCTIONEER	
Description: Mediating between suppliers and bidders in the Auction model.	
Protocols and Activities: ServiceProposed, Offers, ReceivePriceOffers, PriceEvaluation, AcceptPrice, AskForNewBid, Inform	
Permissions: reads offer_definition // the offer made by the seller changes price // the highest proposed price	
Responsibilities	
Liveness: AUCTIONEER = (ServiceProposed. Offers.ReceivePriceOffers. PriceEvaluation.(AcceptPrice AskForNewBid))*	
Safety: ■ number_of_price_proposals >= 1	

Figure 4.4. Schema for role AUCTIONEER

eer role receives a proposal of a service (by means of the ServiceProposed protocol), it then offers (using the Offers protocol) this proposal to the agents fulfilling the bidders role, it receives price offers (using the ReceivePriceOffers protocol) from the set of bidding agents, and then may accept the price or ask for a new bid, using the AcceptPrice or AskForNewBid protocols respectively. The safety expression states that an agent playing the Auctioneer role needs at least one price proposal.

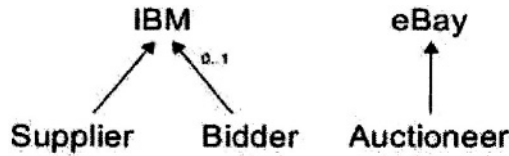


Figure 4.5. A sample agent model

In addition to the role model, the Gaia analysis phase includes the definition of an interaction model, including a protocol definition for each protocol of each role in the system. More attention is paid to the nature and purpose of the interaction than to the sequence of execution steps and message exchanges. In fact, the protocol definition describes the high-level purpose of the protocol, ignoring implementation details such as the sequence of messages exchanged. In particular, the protocol definition is a simple table detailing the role initiating the protocol, the role in charge of responding to it, the input and output information processed in the protocol, as well as a brief textual description of the type of information processing taking place during the execution of this protocol.

2.2 Design with Gaia

In the design phase, the abstract constructs of the analysis stage, i.e., the roles and protocols represented in the role and interaction models, are mapped into concrete constructs, i.e., the agent types that will be instantiated at runtime.

Gaia requires three models to be produced in the design phase (see Figure 4.1): the agent model specifying the types of agents to form the actual system, the service model specifying the services to be implemented by these agent types, and an acquaintance model depicting communication links between agent types.

Assigning roles to agent types creates the agent model. Each agent type may be assigned to one or more roles. For each agent type, the designer annotates the cardinality of agent instances of that type at runtime. Figure 4.5 shows a sample agent model where IBM takes both the Supplier role and the Bidder role.

In Gaia, a service is simply a coherent block of functionality, neutral with respect to implementation details. The service model lists services that agent types provide. The services are derived from the activities and protocols of the roles. For each service, four attributes must be specified, namely the inputs, outputs, pre-conditions and post-conditions. They are easily derived from attributes such as protocol input, from the role model and the interaction model.

The acquaintance model is a directed graph between agent types. An arc from A to B signals the existence of a communication link allowing A to send messages to B. The purpose is to allow the designer to visualize the degree of coupling between agent types. In this model, further details such as message types are ignored.

2.3 Limitations

Gaia was designed to handle small-scale, closed agent-based systems. Consequently, it has weaknesses that render it inappropriate for engineering complex open systems like agent marketplaces. Specifically, Gaia has the following limitations:

- 1 Gaia cannot explicitly model and represent important social aspects of a MAS. Among them, Gaia cannot explicitly model the organizational structure of the agents in the system, or alternatively, the architecture of the system with merely non-recursive roles. It also lacks the ability to explicitly model the social goals, social tasks or organizational rules within an organization of agents. These factors, together with the fact that Gaia implicitly assumes all agents to be cooperative, make it clearly unsuitable for open agent systems.
- 2 Gaia employs Gaia-specific notations for representing roles and protocols. These notations – although simple and easy to catch – may be somewhat poor for expressing complex problems such as complex multi-phase interaction protocols.

Further limitations can be identified which are discussed less extensively in this chapter. Gaia lacks a requirements modeling phase. Gaia lacks appropriate environmental modeling, and domain knowledge modeling.

3. Gaia v.2

The official extension of Gaia, to which we will refer here as Gaia v.2, extends Gaia based on the key consideration that an organization is more than a simply a collection of roles, as was considered in the first version of Gaia. Additional organizational abstractions are to be identified (Zambonelli et al., 2003).

In particular, in Gaia v.2, in addition to roles and protocols, the environment in which a MAS is immersed is elected to a primary analysis and design abstraction. The environment abstraction explicitly specifies all the entities and resources a MAS may interact with to reach the organizational goal. In the original version of Gaia, the description of the environment was implicit in the definition of the permissions associated with roles, a choice that does not

promote a clear understanding of the overall system and that does not help to capture interactions between agents that may occur via the mediation of the environment.

In addition to the environmental model, two further organizational abstractions come into play in Gaia v.2, namely, the organizational rules and the organizational structures.

Organizational rules aim to specify some constraints that the organization will have to observe. Organizational rules may be global (concerned with all the roles and protocols), or just concerned with the relations between some roles, between some protocols, or between some roles and protocols in the MAS. Organizational rules allow the system designer to explicitly define when and under which conditions a new agent may participate in the organization, which is its position, as well as which behaviors are accepted as self-interested expressions and which ones have instead to be prevented by the organization.

Organizational structures, on the other hand, aim to make explicit the overall architecture of the systems (the position of each role in the organization and its relationship with other roles) and its choice, a choice that was instead implicitly defined by the role model in the original Gaia.

Organizational rules and organizational structures are strictly related, in that organizational rules may help designers in the identification of organization structures that more naturally suit these rules. In this sense, Gaia v. 2 recognizes that the role model should explicitly derive from the choice of the organizational structure, and not vice versa.

Considering all the above, an overview of the Gaia v.2 methodology with its models and their relationships is presented in Figure 4.6.

3.1 Analysis in Gaia v.2

The analysis phase includes the identification of:

- The goals of the organizations that constitute the overall system and their expected global behavior. At this step it is important to identify useful decomposition of the global organization into sub-organizations.
- The environmental model that represents the environment (in terms of computational variables/resources) in which the MAS will be situated.
- The preliminary roles model. As in the original version of Gaia, the notion of roles abstracts from any mapping into agents (this issue will be considered in the design phase). However, in Gaia v.2, the analysis has to avoid the imposition of a specific organizational structure implicitly defined via the role model. Instead, Gaia v.2 prescribes to leave the role model incomplete (i.e., with some of the inter-role interactions not identified), since only an accurate identification of the organizational struc-

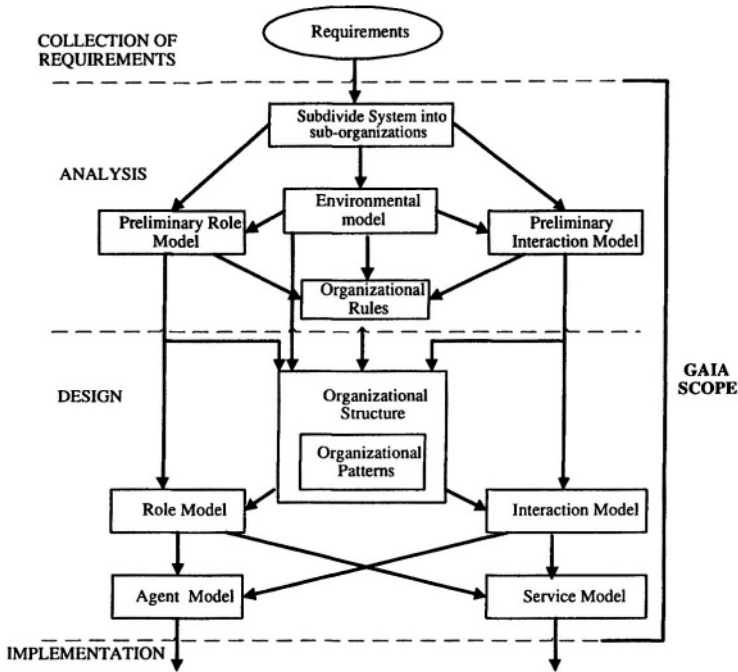


Figure 4.6. Models of Gaia v.2 and their Relationships (Zambonelli et al., 2003)

ture – to take place in the design phase – will enable exact understanding of which roles will interact with which others.

- The preliminary interaction model, which, as in the case of the role model, abstracts away from the organizational structure and has thus to be preliminary (e.g., with some of the partners in a protocol undefined).
- The organizational rules that govern the organization in its global behavior. Such rules impose constraints on the execution activities of roles and protocols. They are fundamental to efficiently specify how, in an open MAS, external self-interested agents can execute without undermining the overall consistency of the developing MAS.

The output of the analysis phase consists of four basic models: (i) the environmental model; (ii) a preliminary roles model; (iii) a preliminary interactions model; and (iv) a set of organizational rules.

3.2 Design in Gaia v.2

The design phase includes the following sub-phases:

- Definition of the overall architecture of the system, i.e., of the organizational structure, taking care that it accommodates all preliminary roles and interactions identified in the analysis phase, and taking care that the adopted structure facilitates the enactment of the organizational rules.
- A great number of different organizational structures may be available for designers to better deal with functional and efficiency requirements. Nevertheless, it is highly probable that a reduced subset of these structures are normally adopted. This opens up the opportunity to exploit, in this phase, existing catalogs of organizational patterns.
- Revision and completion of the preliminary role and interaction models, on the basis of the adopted organizational structure.
- Definition of the agent model specifying agent types (a set of agent roles) and agent instances, as in the original version of Gaia.
- Definition of the services model, as in the original version of Gaia, to specify the main services (blocks of activities with their pre-conditions and post-conditions) that agent types have to provide.

3.3 Discussion

The general framework of Gaia v.2 exploits consistently novel organizational abstractions to overcome some of the limitations identified in the original version of Gaia. Specifically, Gaia v.2 is more oriented to designing and building systems in complex, open environments. It is easy to see, in fact, that the explicit adoption of an environmental model, of the organizational rules and of the organizational structures, enables capturing and modeling the agent marketplace example in a more effective and flexible way.

The need to request goods and/or a service usually implies, in agent marketplaces, the request for a set of offers by sellers, and receipt of the offers, and the evaluation by the buyer, after which the service provision is assigned to the winner. However, the choice of which specific process to adopt for this transaction represents an important design choice, that should be made explicit, as in Gaia v.2, and that requires a proper identification of the environmental characteristics and of the organizational rules.

In a closed system, making buyer and sellers interact directly, without the mediation of any auctioneer, may be satisfactory since not self-interested behavior is likely to occur. However, in an open system, specific organizational rules may be identified that should govern the interactions between roles and that should drive the identification of the organizational structures. If we assume that agents are not benevolent to each other (as in real-world open auctions), the need to properly constrain the way agents negotiate (e.g., via a rule

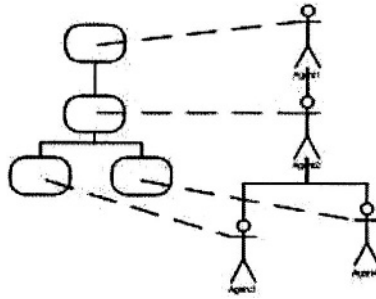


Figure 4.7. System viewed as a computational organization made up of the role hierarchy and the agent hierarchy

requiring buyers to submit bids to sellers in an ordered and monotonically increasing way) may require adopting an organizational structure in which a central role is given to an agent playing the Auctioneer role. Also, the explicit representation of the environment (i.e., a computational environment made up of goods, prices, bids) may enable identifying some problems such as, e.g., the fact that specific bids should not be made public or should not be changed arbitrarily by agents.

4. The ROADMAP Methodology

Another extension to Gaia is ROADMAP, proposed at the University of Melbourne and first described in (Juan et al., 2002). It should be noted that ROADMAP was proposed earlier than Gaia v.2. Coherency of presentation dictated the discussion of Gaia v.2 before ROADMAP in this chapter.

ROADMAP started as an attempt to extend the original version of Gaia with: a dynamic role hierarchy (as a way to deal with open agent systems), additional models to explicitly describe the agent environment (as Gaia v.2 does), and the agent knowledge (a feature that is very important in intelligent agent systems and that is neglected by both Gaia and Gaia v.2). From Gaia, ROADMAP inherits – other than the basic underlying process model – the organizational view on MAS, and the basic definitions of roles, protocols, agents and services. However, over time the semantics of these concepts in ROADMAP has become quite different from Gaia, so that ROADMAP can, to some extent, be considered a methodology on its own.

In ROADMAP, a system is viewed as an organization of agents, consisting of a role hierarchy and an agent hierarchy (Figure 4.7).

The role hierarchy is the specification of the system, representing the correct behaviour of agents. The agent hierarchy is the implementation of the system, providing the actual functionalities. The role hierarchy constrains the agent

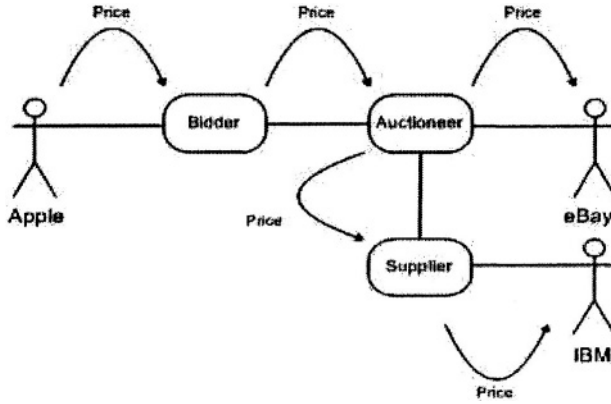


Figure 4.8. Message passing between agents via their roles

hierarchy in the same way as organizational structures, responsibilities and business procedures constrain individuals in a human organization. To some extent, the role hierarchy plays in ROADMAP a similar role that organizational structures and organizational rules play altogether in Gaia v.2.

Roles and protocols, as in Gaia, are first class entities in ROADMAP that, unlike in Gaia, have concrete runtime realization in ROADMAP. In an organization of ROADMAP, in fact, agents interact by message passing, while roles and protocols act as message filters (Figure 4.8).

Figure 4.8 shows an example of an official interaction in an organization between Agent A and Agent B. The message from Agent A is first sent to and validated by its role. If all constraints are satisfied, the message propagates to Agent B’s role. After the message is validated, Agent B receives the message and can now respond to it. As part of the organizational arrangement, the message is also forwarded to Agent C’s role and to Agent C after validation for monitoring purpose. If the message fails to satisfy constraints from any roles concerned, the message will be rejected and actions will be taken to handle the error. This mechanism ensures the interaction respects perspectives of all roles involved. Direct message passing between agents are considered private and does not have the same official status in the organization. In some organizations, private interaction is not desirable and maybe forbidden.

Protocols are reusable message patterns. As concrete runtime entities, they can be reasoned and manipulated, effectively re-routing the interaction without affecting the agent services (in grey) behind the protocols (Figure 4.9). ROADMAP combines this notation with AUML (see chapter 12) to overcome the rather poor notation for protocols adopted by Gaia and inherited by Gaia v.2.



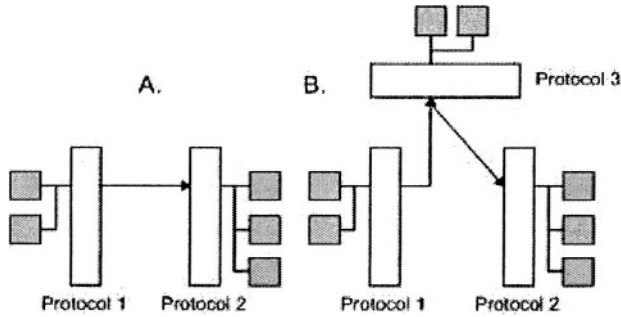


Figure 4.9. Re-routing interaction with protocols

Figure 4.10 shows an example ROADMAP role. The main improvements from Gaia roles are:

- 1 The new Sub-Roles attributes that use the aggregation semantics for building a role hierarchy recursively. The “involves” keyword relates sub-role attributes to parent role attributes. For example, a parent role safety condition is maintained if and only if all involved sub-role safety conditions are maintained.
- 2 The new knowledge attributes associating knowledge components with roles.
- 3 Use of keywords “before”, “during” and “after” to limit the applicability of attributes to a liveness state or a protocol. This allows pre-conditions, post-conditions and invariants of protocols to be defined and the implementing services constrained at runtime.
- 4 Evaluation functions such as Profit_Margin are specified. The functions serve as an official measure of agent performance. The Goals attributes nominate the correct evaluation functions for agents to optimize at a given state. These functions can be implemented in any roles, agents or resources.
- 5 The permission attribute can now include read or modify access to other roles or protocols, allowing the organization to be changed at runtime given the proper authorization.

The key ROADMAP concepts are outlined in the ROADMAP meta-model (see Figure 4.11)

Figure 4.12 shows the structure of the ROADMAP models. The models are grouped into three categories. The environment model and the knowledge

Role Schema: Supplier
Description: The role of proposing a service and/or good in the "Auction" model (the role of the agents class Sellers).
Sub-Roles: Marketing, Sales, ServiceStaff //sub-roles
Knowledge: Sales History //history of previous sales Market Trend //knowledge on the current market demands
Responsibilities: Liveness: Supplier = (sellService Preparation)* sellService = ServiceProposal . ReceiveApproval Safety: 1. Profit_Margin(offer_definition) >= 10% during ServiceProposal 2. Customer_Awareness (offer_definition) >= 30 % during Preparation involves Marketing . Safety1
Goals: ProfitGoal = Max. Profit_Margin during ServiceProposal CusGoal = Max. Customer_Awareness during Preparation involves Marketing . CusGoal According to Prioritize ()
Protocol and Activities: ServiceProposal involves Sales . MakeProposal, ReceiveApproval involves Sales . ReceiveApproval, Preparation involves Marketing . Research and ServiceStaff . CreateService
Permissions: Changes offer_definition //service and/or good proposed Changes Market Trend during Preparation //modify a knowledge component Changes This_Role . Protocols during Preparation //allow reflection on all protocols of //this role; self-modification

Figure 4.10. A sample ROADMAP role definition

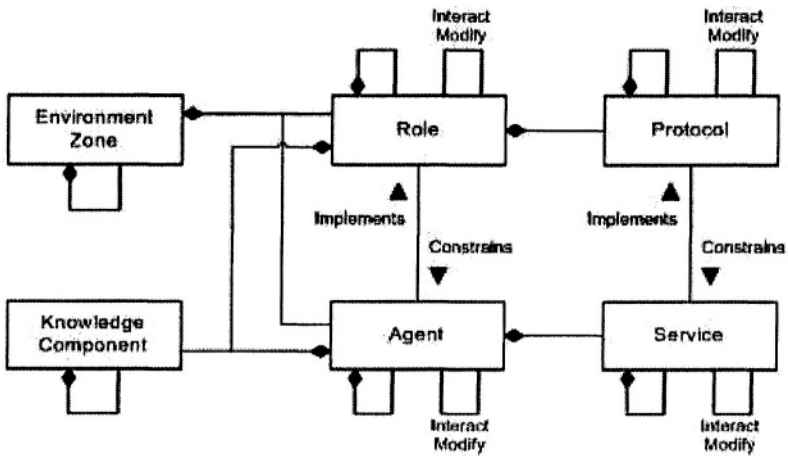


Figure 4.11. The ROADMAP meta-model

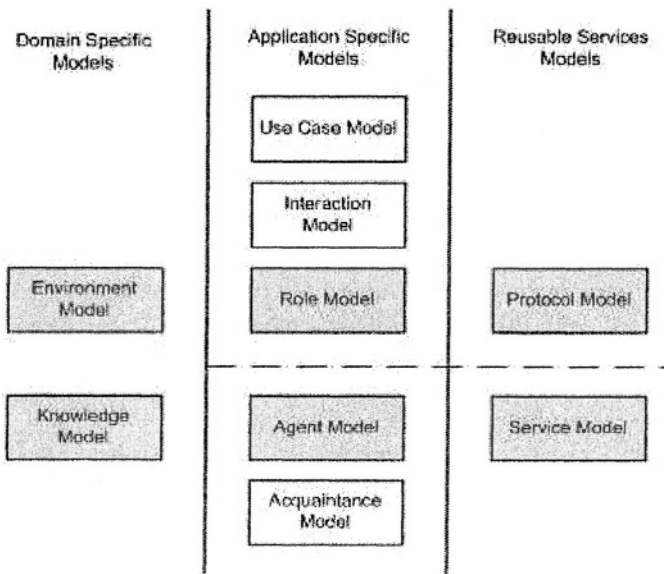


Figure 4.12. Models in the ROADMAP methodology

model contain reusable high-level domain information. The use-case model, interaction model, role model, agent model and acquaintance model are application specific. The protocol model and service models describe potentially reusable low-level software components.

5. Extending Gaia with AUML

Gaia notation for representing roles, protocols, and multiagent organizations in their whole is quite poor and unlikely to be widely accepted for industry solutions. The notation has not been substantially enriched or extended in Gaia v.2. The gap from industrial practice is quite evident with respect to the specification of agent interactions, as pointed out in (Shehory and Sturm, 2001). In effect, the Gaia protocol model notation considers all the relevant aspects of a protocol but may be too extensive to specify (one model for every interaction). Further, the notation is quite informal and not based on a standard accepted by industry. Although ROADMAP partially overcomes these problems by introducing richer notations, the possibility to adhere to existing standard notations has to be evaluated.

5.1 AUML – Key Concepts

Our proposed extension is to re-use Agent UML (AUML) (see chapter 12), a set of extensions to UML notation that have been proposed for modeling agent-based systems. AUML builds on the acknowledged success of UML in supporting industrial-strength software engineering. The core part of AUML is the Agents Interaction Protocol (AIP). Protocols in AIP are specified by means of protocol diagrams (extended sequence diagrams) that allow designers to specify extended message semantics, parameterized nested protocols, and protocol templates.

The key ideas of AUML that may be integrated into Gaia to enrich its expressiveness for specifying agent interactions are:

- 1 The protocol can be regarded as a whole entity and treated as a package. AUML considers an AIP as a template, whose parameters may be roles, constraints, and communication acts. This template approach expresses in a more compact way and UML-like notation the same semantics of the Gaia protocol notation, but it is easier to visualize.
- 2 Each protocol implies inter-agent interactions that are described using sequence diagram, activity diagrams, and statecharts. AUML extends sequence diagram notation in order to represent Agents (and eventually their Class) and their Roles, and to support concurrent threads of interactions. The activity diagram, particularly useful for complex interaction protocols that involve concurrent processing, and statecharts are used to specify the internal behavior of an agent.

AUML proposes other extensions to UML in order to better capture richer role specification, packages with agent interfaces, deployment diagrams indicating mobility, emergence, etc. However, those notations are less rich than

those proposed for AIP and some of them are poor compared with Gaia notations. For example, role specification in Gaia is more expressive, formal and includes more relevant aspects (permissions and responsibilities) than proposed in AUML.

Moreover, AUML is not a thorough methodology and does not cover all the abstraction proposed by Gaia v.2. Specifically, AUML offers a quite poor notation in covering the organizational structures and does not consider the organizational rules (Parunak and Odell, 2001). It presents some barriers to adapt to complex and open systems with self-interested behavior.

5.2 The Gaia Interactions Model in AUML

The proposed notation of Gaia for protocols is quite informal. Thus, instead of the Gaia notation, we introduce the use of AIP notation proposed by AUML. This implies that a protocol must be described as a sequence of actions and message interactions and may contain a set of atomic protocols as defined in Gaia. In the agents marketplace example we have considered two protocols for the contract phase: one for the “Wanted Request” model and one for the “Auction” model. For space reasons we present just the package and the activity diagrams for the “Auction” protocol, avoiding the statechart.

In Figure 4.13 and 4.14 it is possible to observe the protocol specifying an Auction model with its respective activity diagram. It states that when the Supplier proposes a service, the Auctioneer offers it to the Bidder looking for the best price. Meanwhile, a Bidder may inform the Auctioneer that he has not understood the proposal or may present a price offer. Once the Auctioneer has evaluated the price it may accept it informing the Supplier, or reject it asking for new bid.

The integration of AUML within Gaia leads to a richer notation for the specification of protocols and inter-agent interactions since the AUML notation introduces different advantages. First, it specifies a distinguished set of agent instances satisfying the agent role and class it belongs to; while Gaia just specifies the role. Second, it is more compact specifying in a single diagram a sequence of actions and messages interactions which may contain a set of atomic protocols as defined in Gaia. Third, AUML is more formal and allows the specification of time ordering of messages between agents. Finally, AUML notation introduces an opportunity for agents to select a path in the interaction according to their goals. The latter two aspects are described in Gaia using natural language, introducing possible ambiguities and misunderstandings.

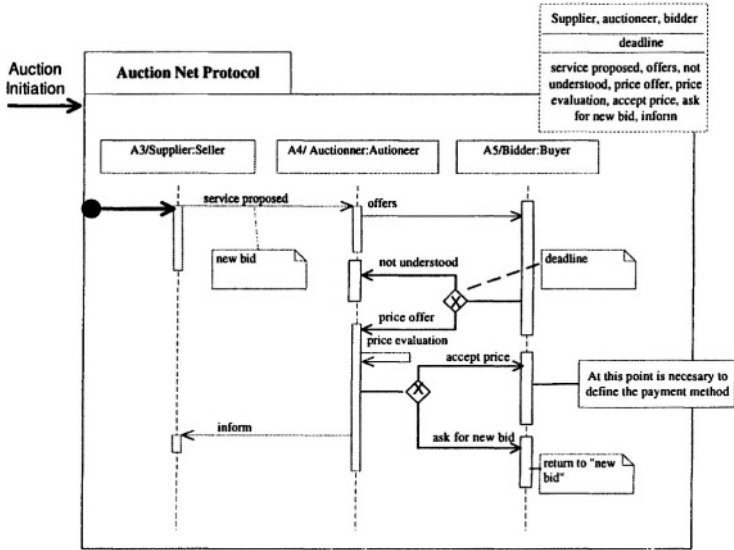


Figure 4.13. The auction model protocol

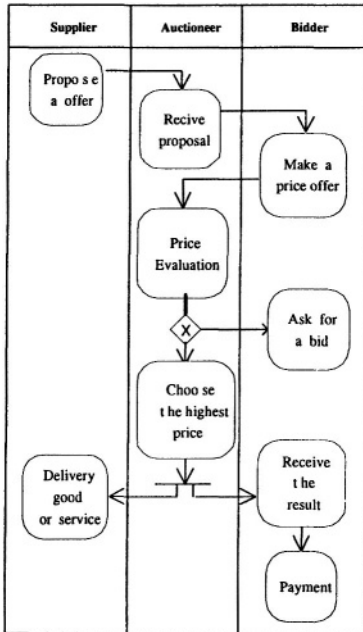


Figure 4.14. The auction model protocol activity diagram

6. Open Issues

Although the above described extensions are important steps towards the improvement of the Gaia methodology, several other directions for improvements can be identified.

One of these directions related to the modeling and specification of the MAS environment. To model the environment, Gaia uses a notation inspired by FUSION for operation schemata (Coleman et al., 1994) complemented (in Gaia v.2) by a graphical representation of the spatial, physical, or logical relationships between environment resources and agents. Other authors give increasing importance to environment modeling, which may be captured using traditional object diagrams, e.g., (Parunak and Odell, 2001). However, in most of the cases (as well as in Gaia) these representations fail in properly capturing the dynamic aspects of the environment. MAS environments may have their own dynamics, which can notably influence the execution of a MAS, and such dynamics deserve specific modeling for the analysis and design of a MAS to be effective.

In addition, the organizational structures in Gaia are currently modeled using a simple, non standard, notation, simply expressing relationships among roles (or agents) and complemented by graphical representations and textual comments aimed at enriching the semantics. These representation, as well as the representation of the organizational rules (currently exploiting FUSION-like notation and temporal logic), may be improved.

Another important limitation refers to the lack, in Gaia, of any requirements modeling phase. Although ROADMAP goes in that direction by proposing the use of use case model to capture functional requirements, this is not enough. For instance, specific agent methodologies exist that emphasise the requirements modeling phase. Among them, Tropos (see chapter 5) seems to be the most formal, complete, and consistent. In the future, it may be useful to explore the opportunity of exploiting and integrating in Gaia the models and notations already exploited in Tropos for requirements engineering.

7. Conclusions

This chapter has summarized the key characteristics of the original version of the Gaia methodology, and has presented three extensions that have been proposed to it in order to overcome its limitations. On the one hand, ROADMAP and the second version of Gaia, Gaia v.2, extend the original methodology with additional abstractions that are necessary to make Gaia suitable for the analysis and design of complex open agent systems. On the other hand, a proposal to integrate the standard AUML notation in the Gaia process can make Gaia more expressive and easier to be accepted by software engineers.

Although the authors consider Gaia – when enriched with the extensions described in this chapter – as one of the most effective methodologies for the analysis and design of MAS, they are also aware of several limitations currently affecting it. For instance, the lack of a requirements modeling phase and the need for adopting even richer and expressive notations than Gaia and AUML currently provide may require further research work.

Acknowledgments

The second author would like to acknowledge the support of the Smart Internet Technology CRC.

Chapter 5

THE TROPOS METHODOLOGY

An Overview

Paolo Giorgini, Manuel Kolp, John Mylopoulos and Marco Pistore

Abstract The objective of this chapter is to give an overview of Tropos methodology. Tropos is based on two key ideas. First, the notion of agent and related mentalistic notions, such as goals and plans, are used in all phases of software development, from early analysis down to the actual implementation. Second, Tropos covers the very early phases of requirements analysis, thus allowing for a deeper understanding of the environment where the software-to-be will eventually operate. We illustrate the phases of the methodology, the Formal Tropos language, and the social and intentional models that are used to support software development.

1. Introduction

The explosive growth of application areas such as electronic commerce, enterprise resource planning, and peer-to-peer computing has deeply and irreversibly changed our views on software and Software Engineering. Software must now be based on open architectures that continuously change and evolve to accommodate new components and meet new requirements. Software must also operate on different platforms, without recompilation, and with minimal assumptions about its operating environment and its users. As well, software must be robust and autonomous, capable of serving end users with a minimum of overhead and interference. These new requirements, in turn, call for new concepts, tools and techniques for engineering and managing software.

For these reasons – and more – agent-oriented software development is gaining popularity over traditional development techniques, including structured and object-oriented ones, see, e.g., (Jennings, 2000). After all, agent-based architectures *do* provide for an open, evolving architecture that can change at run-time to exploit the services of new agents, or replace under-performing ones. In addition, software agents can, in principle, cope with unforeseen cir-

cumstances because their architecture includes goals along with a planning capability for meeting them.

We are currently working on an agent-oriented development methodology called Tropos (Castro et al., 2002). In a nutshell, Tropos is based on two key features. First, the notion of agent and related mentalistic notions are used in all software development phases, from the early requirements analysis down to the actual implementation. Second, the methodology emphasizes early requirements analysis, the phase that precedes the prescriptive requirements specification. In this respect, Tropos is quite different from other agent- and object-oriented software development methodologies.

Paying attention to the activities that precede the specification of prescriptive requirements for the system-to-be (Dardenne et al., 1993; Yu, 1995) means that developers can capture and analyze the goals of stateholders. These goals play a crucial role in defining the requirements for the new system. Put another way, prescriptive requirements capture the *what* and the *how* for the system-to-be. Early requirements, on the other hand, capture the reasons *why* a software system is developed. This new perspective, in turn, supports a more refined analysis of system dependencies and a more uniform treatment of functional and non-functional requirements.

Tropos adopts Eric Yu's *i** model which offers actors (agents, roles, or positions), goals, and actor dependencies as primitive concepts for modeling an application during early requirements analysis. Tropos is intended to support four phases of software development: *early requirements* analysis, concerned with the understanding of a problem by studying its organizational setting; *late requirements* analysis, where the system-to-be is described within its operational environment, along with relevant functions and qualities; *architectural design*, where the system's global architecture is defined in terms of subsystems, interconnected through data, control, and other dependencies; and *detailed design*, where the behavior of each component is defined in further detail.

The objective of this chapter is to present the Tropos methodology. Section 2 offers an overview of the methodology, while section 3 presents the Tropos formal language, designed to support the methodology. Section 4 describes the social patterns used during the development process, while section 5 presents its goal model. Finally, section 6 summarizes the contributions of the proposed methodology and points to directions for further work.

2. Overview

In this section we present briefly the four phases supported by Tropos, using the *Media Shop* case study. *Media Shop* is a store selling and shipping media items such as books, magazines, audio CDs, videotapes, and the like. *Media Shop* customers (on-site or remote) can use a catalogue describing available

items to fill their orders. *Media Shop* is supplied with the latest releases from *Media Producer* and in-catalogue items by *Media Supplier*. To increase market share, *Media Shop* has decided to open up *Medi@*, a B2C internet site. Through it, a customer can put in orders to *Media Shop* through the internet. She can also search the on-line store by either browsing the catalogue, or by querying the database through keywords or full-text search. The system uses communication facilities provided by *Telecom Cpy* and on-line financial services supplied by *Bank Cpy*.

Early Requirements Analysis. It focuses on the intentions of stakeholders. Intentions are modeled as goals. Through some form of goal-oriented analysis, these initial goals eventually lead to the functional and non-functional requirements of the system-to-be (Dardenne et al., 1993). In *i** (Yu, 1995), stakeholders are represented as (social) actors who depend on each other for goals to be achieved, tasks to be performed, and resources to be furnished. The *i** framework includes the *strategic dependency model* for describing the network of relationships among actors, as well as the *strategic rationale model* for describing and supporting the reasoning that each actor goes through concerning its relationships with other actors.

A strategic dependency model is a graph involving *actors* who have *strategic dependencies* among each other. A dependency describes an “agreement” (called *dependum*) between a depending actor (*dependor*) and an actor who is depended upon (*dependee*). The type of the dependency describes the nature of the agreement. *Goal* dependencies are used to represent delegation of responsibility for fulfilling a goal; *softgoal* dependencies are similar to goal dependencies, but their fulfillment cannot be defined precisely (for instance, the degree of fulfillment is subjective); *task* dependencies are used in situations where the dependee is required to perform a given activity; and *resource* dependencies require the dependee to provide a resource to the dependor. As shown in Figure 5.1, actors are represented as circles; dependums – goals, softgoals, tasks and resources – are respectively represented as ovals, clouds, hexagons and rectangles; and dependencies have the form *dependor* → *dependum* → *dependee*.

These elements are sufficient for producing a first model of an organizational environment. For instance, Figure 5.1 depicts an *i** model of our *Medi@* example. The main actors are *Customer*, *Media Shop*, *Media Supplier* and *Media Producer*. *Customer* depends on *Media Shop* to fulfill her goal: *Buy Media Items*. Conversely, *Media Shop* depends on *Customer* to *increase market share* and make “*customers happy*.” Since the dependum *Happy Customers* cannot be defined precisely, it is represented as a softgoal. The *Customer* also depends on *Media Shop* to *consult the catalogue* (task dependency). Furthermore, *Media Shop* depends on *Media Supplier* to supply media items in a continuous way and get a *Media Item* (resource dependency). The items are expected to

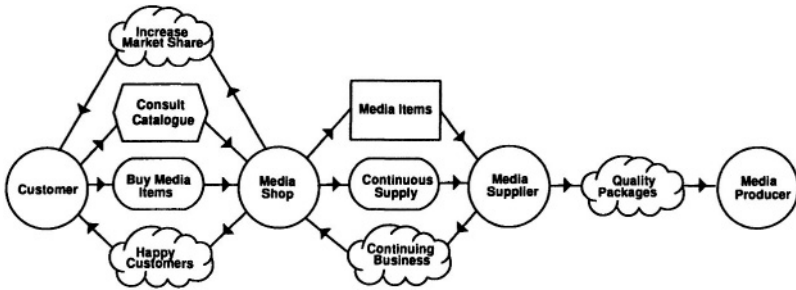


Figure 5.1. *i** Model for a media shop

be of good quality, otherwise the *Continuing Business* dependency might not be fulfilled. Finally, *Media Producer* is expected to provide *Media Supplier* with *Quality Packages*.

Late Requirements Analysis. It results in a requirements specification which describes all functional and non-functional requirements for the system-to-be. In Tropos, the system is represented as one or more actors which participate in a strategic dependency model, along with other actors from the system's operational environment. In other words, the system comes into the picture as one or more actors who contribute to the fulfillment of stakeholder goals.

As late requirements analysis proceeds, the system (*Medi@*) is given additional responsibilities, and ends up as the dependee of several dependencies. A strategic rationale model determines through a means-ends analysis how the system goals (including softgoals) identified during early requirements can actually be fulfilled exploiting the contributions of other actors. A strategic rationale model is a graph with four types of nodes – *goal*, *task*, *resource*, and *softgoal* – and two types of links – means-ends links and decomposition links. A strategic rationale graph captures the relationship between the goals of each actor and the dependencies through which the actor expects these dependencies to be fulfilled.

The analysis in Figure 5.2 focuses on the system itself and postulates a root task *Internet Shop Managed* providing sufficient support (++) to the softgoal *Increase Market Share*. That task is firstly refined (via decomposition links) into goals *Internet Order Handled* and *Item Searching Handled*, softgoals *Attract New Customer*, *Security*, *Adaptability* and *Availability*, and task *Produce Statistics*. To manage internet orders, *Internet Order Handled* needs to be achieved (means-ends link) through the task *Shopping Cart*. In turn, this task is decomposed into subtasks *Select Item*, *Add Item*, *Check Out*, and a subgoal *Get Identification Detail*. These are the main process activities required to design an operational on-line shopping cart. The latter goal is achieved either through secure or standard form orderings.

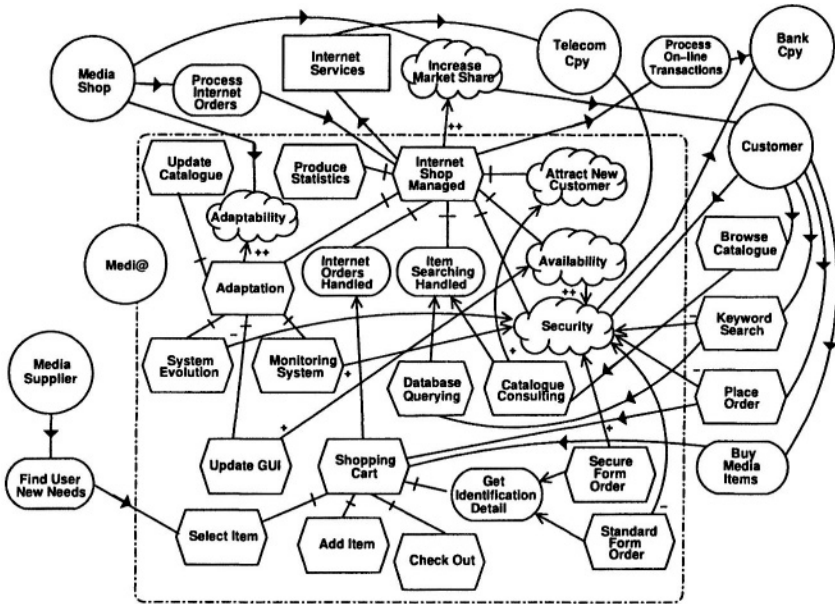


Figure 5.2. Strategic rationale model for Medi@

In addition, Figure 5.2 introduces softgoal contributions to model sufficient and partial positive (respectively ++ and +) or negative (respectively -- and -) support to softgoals *Security*, *Availability*, *Adaptability*, *Attract New Customers* and *Increase Market Share*. The result of such a means-ends analysis is a set of (system and human) actors who are dependees for some of the dependencies that have been postulated.

Resource, task and softgoal dependencies correspond naturally to functional and non-functional requirements. Leaving (some) goal dependencies between system actors and other actors is a novelty. Traditionally, functional goals are “operationalized” during late requirements, while quality softgoals are either operationalized or “metricized” (Dardenne et al., 1993). In our example, we have left four (soft)goals (*Availability*, *Security*, *Adaptability* and *Increase Market Share*) for architectural design. The operationalization of these non-functional requirements will depend on the type of architecture chosen during design.

Architectural Design. A system architecture constitutes a relatively small, intellectually manageable model of system structure, which describes how system components work together (Shaw and Garlan, 1996). In Tropos, we have defined organizational architectural styles (Kolp et al., 2001) for cooperative, dynamic and distributed applications – such as MAS – to guide the design of the system architecture. These organizational architectural styles are based on

concepts and design alternatives coming from research in organization management. As such, they help match a MAS architecture to the organizational context within which the system will operate. We present more details on these organizational styles and the Tropos architectural design phase in section 4.

Detailed Design. It introduces additional detail for each architectural component of a system. In particular, this phase determines how the goals assigned to each actor are fulfilled by agents in terms of design patterns. Design patterns, e.g., (Gamma et al., 1995), have attracted much attention. Unfortunately, the literature focuses on object-oriented patterns, rather than the intentional and social ones that are relevant here. Within Tropos, social patterns (Do et al., 2003) are used to find a solution to a specific goal defined at the architectural level through the identification of organizational styles and relevant quality attributes. More details about social patterns are presented in section 4.

Detail design in Tropos also includes the specification of agent communication and agent behavior. To support this task, we propose to adopt existing agent communication languages, such as FIPA-ACL (Labrou et al., 1999), and extensions to UML, such as the Agent Unified Modeling Language (AUML) (see chapter 12).

3. Formal Tropos

The Tropos framework supports the application of formal analysis techniques for the verification of requirements specifications. The analysis is based on *Formal Tropos* (hereafter FT), a specification language that offers all the standard mentalistic notions of Tropos and supplements them with a rich temporal specification language inspired by KAOS (van Lamsweerde, 2001). FT allows for the description of the dynamic aspects of Tropos models. More precisely, in FT we focus not only on the intentional elements themselves, but also on the circumstances in which they arise, and on the conditions that lead to their fulfillment. In this way, the dynamic aspects of a requirements specification are introduced at the strategic level, without requiring an operationalization of the specification. With an FT specification, one can ask questions such as: Can we construct valid operational scenarios based on the model? Is it possible to fulfill the goals of the actors? Do the dependencies represent a valid synchronization between actors?

In this section we give a short description of the key aspects of the FT language. A full definition can be found in (Fuxman, 2001; Fuxman et al., 2001).

An FT specification describes the relevant elements (actors, goals, dependencies, etc.) of a domain and the relationships among them. The description of each element is structured in two layers. The outer layer is similar to a class declaration. It associates to the element a set of attributes that define its

```

Entity Item
Entity Cart
  Attribute items : set of Item
Actor Customer
Actor Medi@
Goal Dependency PlaceOrder
  Depender Customer
  Dependee Medi@
  Mode achieve
Task ShoppingCart
  Actor Media@
  Mode achieve
  Attribute constant cart : Cart
    constant po : PlaceOrder
Task AddItem
  Actor Student
  Mode achieve
  Attribute constant sc : ShoppingCart
    constant item : Item
Goal GetIdentificationDetail
  Actor Medi@
  Mode achieve
  Attribute constant sc : ShoppingCart
    customer : Customer
SoftGoal Security
  Actor Medi@
  Mode maintain
    
```

Figure 5.3. Excerpt of FT class declaration

structure. The inner layer expresses constraints on the lifetime of the objects, given in a typed first-order linear-time temporal logic.

Figure 5.3 is an excerpt of the outer layer of the IT specification of the *Medi@* example. It focuses on the management of the on-line shopping cart. Actors, intentional elements, and dependencies of the Strategic Rational Model are mapped into corresponding “classes” in the outer layer of FT. Moreover, “entities” (e.g., *Cart* and *Item*) are added to represent the relevant and non-intentional elements of the domain. Several instances of a class may exist during the evolution of the system. For example, different *PlaceOrder* instances may exist for different customers, and several *AddItem* tasks can be done during the management of a *ShoppingCart*.

Each class has an associated list of attributes. Most of the attributes in FT are references to other classes and are used to define the relationships among the different instances of these classes. For example, task *ShoppingCart* refers to the specific *Cart* that is being managed (attribute *cart*) and to the *PlaceOrder* dependency that triggered the management of the *ShoppingCart* (attribute *po*). Moreover, each *Cart* refers to the set of items that have been added to

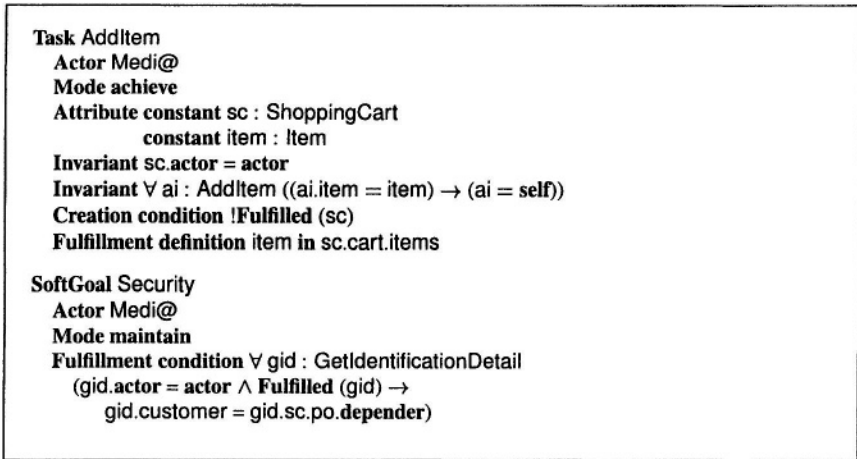


Figure 5.4. Example of FT constraints

it. **Constant** attributes (i.e., attributes whose values do not change over time) define static relations among the class instances of a model. For instance, the cart associated to a given instance of ShoppingCart does not change. The set of items associated to the cart, on the other hand, can change over time. Special attribute **actor** associates a goal or task to the corresponding actor. Similarly, **depender** and **dependee** attributes define the two actors involved in a dependency.

Intentional elements have a **Mode** attribute that defines the modality of the fulfillment of the goal or task. For instance, the mode of task ShoppingCart is **achieve**, which means that the Medi@ actor wants to reach a state where the management of the cart has been fulfilled and the corresponding order has been placed. Softgoal Security, instead, has a **maintain** mode, since the security of the system has to be continuously maintained.

Figure 5.4 contains some examples of constraints on the lifetime of class instances that define the inner layer of an FT specification. **Invariant** constraints define conditions that should be true throughout the lifetime of class instances. Typically, invariants define relations on the possible values of attributes, or cardinality constraints on the instances of a given class. For instance, the first invariant of Figure 5.4 binds an AddItem task with its associated ShoppingCart task, while the second invariant imposes a cardinality constraint on the AddItem tasks for a given Item.

Two critical moments in the lifecycle of intentional elements and dependencies are the instants of their creation and fulfillment. Creation and fulfillment constraints can be used to impose conditions for these two moments in the life of an intentional element. The creation of a goal or task instance means that the owner or depender expects or desires the achievement of the goal/task.

Creation constraints should be satisfied whenever a new instance is created, while **fulfillment** constraints should hold whenever a goal or softgoal is satisfied, a task is performed, a resource is made available, or a dependum is delivered. Creation and fulfillment constraints are further distinguished as sufficient conditions (keyword **trigger**), necessary conditions (keyword **condition**), and necessary and sufficient conditions (keyword **definition**).

A first usage of creation and fulfillment constraints is to relate subordinate goals and tasks with their parent intentional elements. For instance, Figure 5.4 shows that a creation condition for an instance of task AddItem is that the parent task ShoppingCart is not yet fulfilled: it is not possible to add further items to a cart, once an order has been placed and task ShoppingCart has been fulfilled. Together with the fulfillment conditions of task ShoppingCart, this creation condition elaborates the decomposition relation between the two tasks shown in Figure 5.2.

The fulfillment condition of softgoal Security requires that, whenever a GetIdentificationDetail goal has been fulfilled, the customer that has been identified positively (gid.customer) coincides with the customer that is interacting with the system for placing the order (gid.sc.po.**depender**), that is, in a secure system we do not allow for invalid identifications.

Once a FT specification has been defined, it can be formally verified in order to identify errors, ambiguities, and under-specifications. The verification phase usually generates feedback on errors in the FT specification and hints on how to fix them. In order to support the verification process, we have developed a prototype tool, called the T-Tool (Fuxman et al., 2003), that is based on finite-state model checking (Clarke et al., 1999; Cimatti et al., 2002). On the basis of an FT specification, the T-Tool builds a finite model that represents all possible behaviors of the domain that satisfy the constraints of the specification. The T-Tool then verifies whether this model exhibits the desired behaviors.

The T-Tool provides several verification functionalities. *Animation* of the specification consists of an interactive generation of a valid scenario, namely, of a scenario that satisfies all the temporal constraints of the FT specification. Animation allows for an immediate feedback on the effects of constraints and for an early identification of trivial bugs and missing requirements. *Consistency checks* verify that the FT specification is not self-contradictory. Inconsistent specifications occur quite often due to complex interactions among constraints in the specification, and they are very difficult to detect without the support of automated analysis tools. During the consistency checks, the T-Tool verifies that there is some valid scenario that respects all the constraints of the FT specification, that all the goals and dependencies are fulfillable in some scenarios, and other similar properties. *Possibility checks* verify whether we are over-constraining the specification, that is, whether we have ruled out scenarios expected by the stakeholders. These expected scenarios are described in the

FT specification using **possibility** properties. For instance, a scenario that we do not want to rule out is the possibility of interrupting the placement of an order also if we have already added some items to the cart. This property can be expressed by the following FT **possibility**. It requires that, even if items have been added to the cart, it is possible to never fulfill a ShoppingCart task (with “**globally** (c)” we specify that condition c is true through all future history of the model):

Possibility \exists sc: ShoppingCart (sc.cart \neq empty \wedge **globally** (\neg Fulfilled (sc)))

Assertion properties verify whether the requirements are under-specified and allowing for invalid scenarios. Also in this case, **assertion** declarations in the FT specification are used to express conditions on the valid scenarios. For instance, a requirement that one wants to be true is that the system is secure, that is, that softgoal Security is fulfilled:

Assertion \forall sec: Security (Fulfilled (sec))

Since the fulfillment of the security goal depends on the success of goal GetIdentificationDetail, the definition of the fulfillment conditions of this goal need special care. If these conditions do allow for incorrect identifications, the previous assertion is violated and an error is reported during the verification phase.

4. Socially-Based MAS Architectures

System architectural design has been the focus of considerable research during the last fifteen years. This has produced well-established architectural styles and frameworks for evaluating their effectiveness with respect to particular software qualities. Examples of styles are pipes-and-filters, event-based, layered, control loops and the like (Shaw and Garlan, 1996). In Tropos, we are interested in developing a suitable set of architectural styles for MAS. Since the fundamental concepts of a MAS are intentional and social, rather than implementation-oriented, we turn to theories which study social structures that result from a design process, namely *Organization Theory* and *Strategic Alliances*. *Organization Theory*, e.g., (Scott, 1998), describes the structure and design of an organization; *Strategic Alliances*, e.g., (Morabito et al., 1999), models the strategic collaborations of independent organizational stakeholders who have agreed to pursue a set of business goals.

Organization Theory. It describes how organizations are actually structured in practice, offers suggestions on how new ones can be constructed, and how old ones can change to improve effectiveness. To this end, schools of organization theory have proposed models such as the *structure-in-5*, the *pyramid style*, the *chain of values*, the *matrix*, the *bidding style* to try to find and formalize recurring organizational structures and behaviors.

For instance the **structure-in-5**, as Mintzberg proposed (Mintzberg, 1992), specifies that an organization is an aggregate of five sub-structures. At the base level sits the *Operational Core* which carries out the basic tasks and procedures directly linked to the production of products and services (acquisition of inputs, transformation of inputs into outputs, distribution of outputs). At the top lies the *Strategic Apex* which makes executive decisions ensuring that the organization fulfills its mission in an effective way and defines the overall strategy of the organization in its environment. The *Middle Line* establishes a hierarchy of authority between the Strategic Apex and the Operational Core. It consists of managers responsible for supervising and coordinating the activities of the Operational Core. The *Technostructure* and the *Support* are separated from the main line of authority and influence the operating core only indirectly. The Technostructure serves the organization by making the work of others more effective, typically by standardizing work processes, outputs, and skills. It is also in charge of applying analytical procedures to adapt the organization to its operational environment. The Support provides specialized services, at various levels of the hierarchy, outside the basic operating work flow (e.g., legal counsel, R&D, payroll, cafeteria).

Figure 5.5 suggests a possible assignment of system responsibilities for our *Medi@* case study following the structure-in-5 organizational style. It is decomposed into five principal components *Store Front*, *Coordinator*, *Billing Processor*, *Back Store* and *Decision Maker*. *Store Front* serves as the *Operational Core*. It interacts primarily with Customer and provides her with a usable front-end Web application for consulting and shopping media items. *Back Store* constitutes the *Support* component. It manages the product database and communicates to the *Store Front* information on products selected by the user. It *stores* and *backs up* all Web information from the *Store Front* about customers, products, sales, orders and bills to produce *statistical information* to the *Coordinator*. It provides the *Decision Maker* with *strategic information* (analyses, historical charts and sales reports).

The *Billing Processor* is in charge of handling orders and bills for the *Coordinator* and implementing the corresponding procedures for the *Store Front*. It also ensures the secure management of financial transactions for the *Decision Maker*. As the *Middle Line*, the *Coordinator* assumes the central position of the architecture. It ensures the coordination of *e-shopping* services provided by the *Operational Core* including the management of conflicts between itself, the *Billing Processor*, the *Back Store* and the *Store Front*. To this end, it also handles and implements strategies to manage and prevent *security* gaps and *adaptability* issues. The *Decision Maker* assumes the *Strategic Apex* role. To this end, it defines the *Strategic Behavior* of the architecture ensuring that objectives and responsibilities delegated to the *Billing Processor*, *Coordinator* and *Back Store* are consistent with that global functionality.

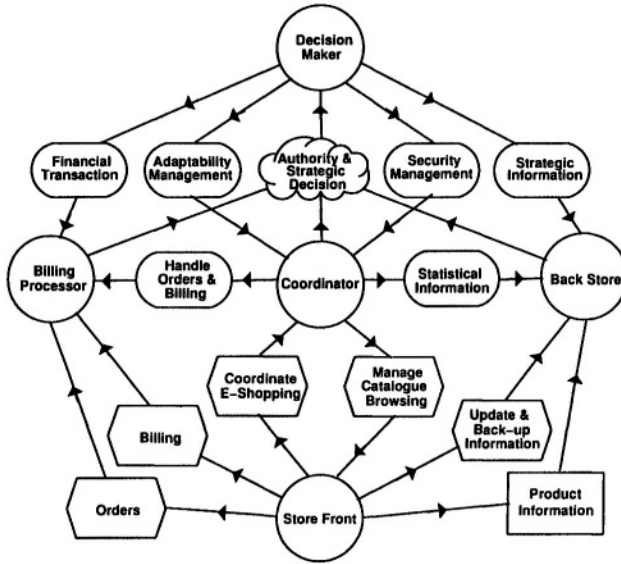


Figure 5.5. The Medi@ organizational architecture in structure-in-5

Strategic Alliances link specific facets of two or more organizations. At its core, this structure is a trading partnership that enhances the effectiveness of competitive strategies of participant organizations by providing for the mutually beneficial trade of technologies, skills, or products derived from them.

For instance, the **joint venture style** involves agreement between two or more intra-industry partners to obtain the benefits of larger scale, partial investment and lower maintenance costs. A specific joint management actor coordinates tasks and manages the sharing of resources between partner actors. Each partner can manage and control itself on a local dimension and interact directly with other partners to exchange resources, such as data and knowledge. However, the strategic operation and coordination of such an organization, and its actors on a global dimension, are only ensured by the joint management actor in which the original actors possess equity participations.

Other styles are the arm's-length style, the hierarchical contracting style or the co-optation style.

Social Patterns. A further step in the architectural design of MAS consists of specifying how the goals delegated to each actor are to be fulfilled (Kolp et al., 2001). For this step, designers can be guided by a catalogue of multi-agent patterns which offer a set of standard solutions. Considerable work has been done in software engineering for defining software patterns, see, e.g., (Gamma et al., 1995). Unfortunately, little emphasis has been put on social and intentional as-

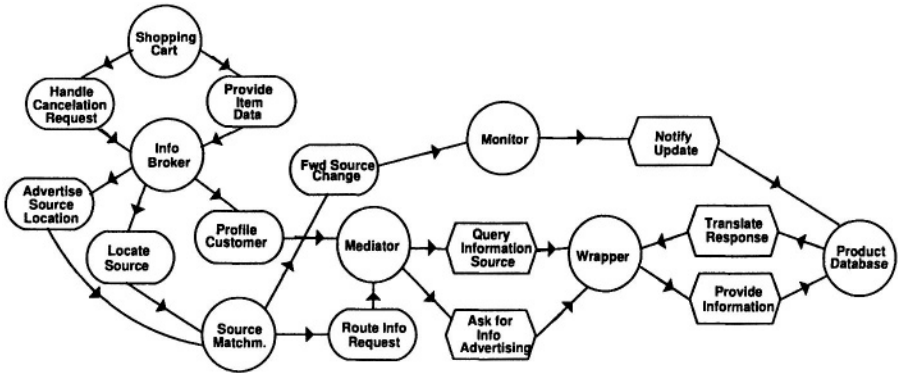


Figure 5.6. Decomposing the store front with social patterns

pects. Moreover, proposals for agent patterns that do address these aspects, see, e.g., (Aridor and Lange, 1998), are not intended for use at a design level. Instead, such proposals seem to aim at the implementation phase, when issues such as agent communication, information gathering, or connection setup are addressed.

Social patterns (Do et al., 2003) are design patterns focusing on social and intentional aspects that are recurrent in multi-agent and cooperative systems. In particular, the structures are inspired by the federated patterns introduced in (Hayden et al., 1999; Kolp et al., 2001). We have classified them into two categories.

The *Pair* patterns – such as booking, call-for-proposal, subscription, or bidding – describe direct interactions between negotiating agents. For instance, the **Bidding** pattern involves an initiator and a number of participants. The initiator organizes and leads the bidding process. He publishes the bid to the participants and receives various proposals. At every iteration, the initiator can accept an offer, raise the bid, or cancel the process.

The *Mediation* patterns – such as monitor, broker, matchmaker, mediator, embassy, or wrapper – feature intermediary agents that help other agents to reach an agreement on an exchange of services. For instance, in the **Broker** pattern, the broker agent is an arbiter and intermediary that requests services from a provider to satisfy the request of a consumer.

Figure 5.6 shows a possible use of the patterns in the e-business system shown in Figure 5.5. In particular, it shows how to realize the dependencies *Manage catalogue browsing*, *Update Information*, and *Product Information* from the point of view of the Store Front. The Store Front and the dependencies are decomposed into a combination of social patterns involving agents, pattern agents, subgoals and subtasks.

The booking pattern is applied between the *Shopping Cart* and the *Information Broker* to reserve available items. The broker pattern is applied to the *Information Broker*, which satisfies the *Shopping Cart*'s requests of information by accessing the *Product Database*. The *Source Matchmaker* applies the matchmaker pattern to locate the appropriate source for the *Information Broker*, and the monitor pattern is used to check any possible change in the *Product Database*. Finally, the mediator pattern is applied to dispatch the interactions between the *Information Broker*, the *Source Matchmaker*, and the *Wrapper*, while the wrapper pattern makes the interaction between the *Information Broker* and the *Product Database* possible.

5. Goal Models

Traditional goal analysis consists of decomposing goals into subgoals by means of an AND- or OR-decomposition. If goal G is AND-decomposed (respectively, OR-decomposed) into subgoals G_1, G_2, \dots, G_n , then all (at least one) of the subgoals must be satisfied for the goal G to be satisfied. Given a goal model consisting of goals and AND/OR relationships among them, and a set of initial labels for some nodes of the graph (S for Satisfied, D for Denied) there is a simple label propagation algorithm which can generate labels for all nodes of the graph (Nilsson, 1971).

Unfortunately, this simple framework for modeling and analyzing goals will not work for many domains where goals cannot be formally defined, and the relationships among them cannot be captured by semantically well-defined relations such as AND/OR ones. For example, a goal such as “*Highly reliable system*” has no formal definition which prescribes its meaning, though one may want to define necessary conditions for its fulfillment. Moreover, such a goal may be related to other goals, such as “*Thoroughly debugged system*,” “*Thoroughly tested system*” in the sense that the latter obviously contribute to the satisfaction of the former, but this contribution is partial and qualitative. In other words, if the latter goals are satisfied, they certainly contribute towards the satisfaction of the former goal, but do not guarantee it. The framework will also not work in situations where there are contradictory contributions to a goal. For instance, we may want to allow for multiple decompositions of a goal G into sets of subgoals, where some decompositions suggest satisfaction of G while others suggest denial.

Tropos proposes a formal model for goals that can cope with qualitative relationships and inconsistencies among goals. Suppose we are modeling the strategic objectives of a US car manufacturer, such as Ford or GM. Examples of such objectives are *increase return on investment* or *increase customer loyalty*. Objectives can be represented as goals, and can be analyzed using goal relationships such as AND, OR, “+” and “-”. In addition, we will use

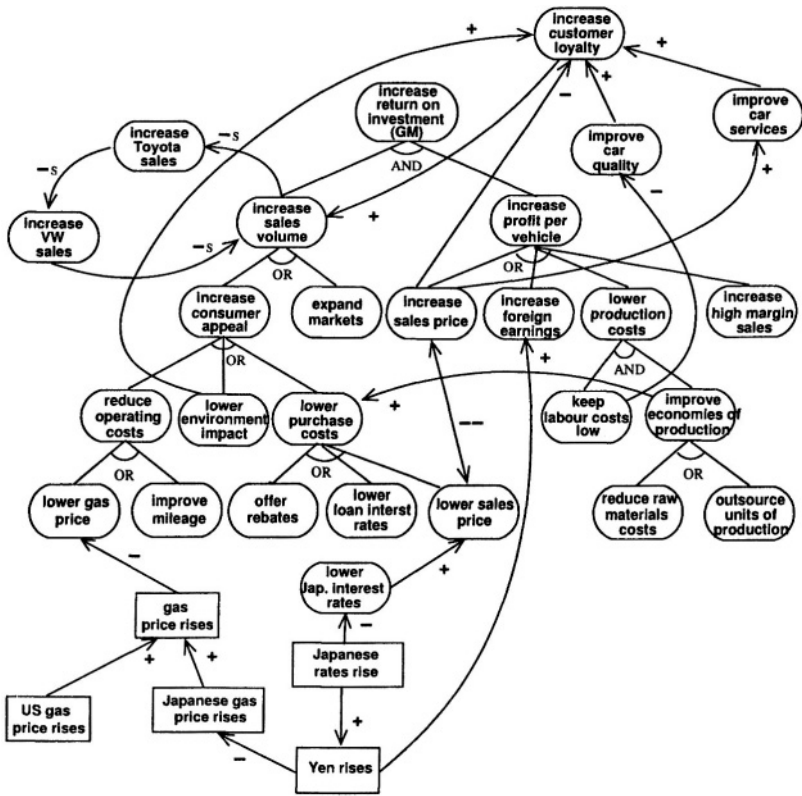


Figure 5.7. A partial goal model for GM

“++” (respectively “--”) a binary goal relationship such that if ++(G, G') (––(G, G')) then satisfaction of G implies satisfaction (denial) of G' .

For instance, *increase return on investment* may be AND-decomposed into *increase sales volume* and *increase profit per vehicle*. Likewise, *increase sales volume* might be OR-decomposed into *increase consumer appeal* and *expand markets*. This decomposition and refinement of goals can continue until we have goals that are tangible (i.e., someone can satisfy them through an appropriate course of action) and/or observable (i.e., they can be confirmed satisfied/denied by simply observing the application domain).

For vaguely stated goals, such as *increase customer loyalty* we may want to simply model other relevant goals, such as *improve car quality*, *improve car services* and relate them through “+” and “-” relationships, as shown in Figure 5.7. These goals may influence positively or negatively some of the goals that have already been introduced during the analysis of the goal *increase*

return on investment. Such lateral goal relationships may introduce cycles in our goal models.

Examples of observable goals are *Yen rises*, *gas prices rise* etc. When such a goal is satisfied, we will call it an *event* (the kind of event you may read about in a news story) and represent it in our graphical notation as a rectangle (see lower portion of Figure 5.7).

Figure 5.7 shows a partial and fictitious goal model for GM focusing on the goal *increase return of investment*. In order to *increase return of investment*, GM has to satisfy both goals *increase sales* and *increase profit per vehicle*. In turn, *increase sales volume* is OR-decomposed into *increase consumer appeal* and *expand markets*, while the goal *increase profit per vehicle* is OR-decomposed into *increase sales price*, *lower production costs*, *increase foreign earnings*, and *increase high margin sales*. Additional decompositions are shown in the figure. For instance, the goal *increase consumer appeal* can be satisfied by satisfying *lower environment impact*, trying to *lower purchase costs*, or reducing the vehicle operating costs (*reduce operating costs*).

The graph shows also lateral relationships among goals. For example, the goal *increase customer loyalty* has positive (+) contributions from goals *lower environment impact*, *improve car quality* and *improve car services*, while it has a negative (−) contribution from *increase sales price*. The root goal *increase return on investment (GM)* is also related with goals concerning others auto manufacturer, such as Toyota and VW. In particular, if GM increases sales, then Toyota loses a share of the North American market; if Toyota increases sales (*increase Toyota sales*), it does so at the expense of VW; finally, if VW increases sales (*increase VW sales*), it does so at the expense of GM.

So far, we have assumed that every goal relationship treats S and D in a dual fashion. For instance, if we have $+(G, G')$, then if G is satisfied, G' is partially satisfied, and (dually) if G is denied G' is partially denied. Note however, that sometimes a goal relationship only applies for S (or D). In particular, the $--$ contribution from increase GM sales to increase Toyota sales only applies when increase GM sales is satisfied (if GM hasn't increased sales, this does not mean that Toyota has). To capture this kind of relationship, we introduce $-_S, -_D, +_S, +_D$ (see also Figure 5.7).

In (Giorgini et al., 2002) we have presented an axiomatization of a qualitative and a quantitative goal model. We report here the qualitative formalization.

We consider sets of goal nodes G_i and of relations $(G_1, \dots, G_n) \xrightarrow{r} G$ over them, including the $(n + 1)$ -ary relations *and*, *or* and the binary relations $+_S, -_S, +_D, -_D, ++_S, --_S, ++_D, --_D, +, -, ++, --$. We briefly recall the intuitive meaning of these relations: $G_2 \xrightarrow{+_S} G_1$ [resp. $G_2 \xrightarrow{++_S} G_1$] means that if G_2 is satisfied, then there is some [resp. a full] evidence that G_1 is satisfied, but if G_2 is denied, then nothing is said about the denial of G_1 ; $G_2 \xrightarrow{-_S} G_1$

[resp. $G_2 \xrightarrow{\neg\mathcal{S}} G_1$] means that if G_2 is satisfied, then there is some [resp. a full] evidence that G_1 is denied, but if G_2 is denied, then nothing is said about the satisfaction of G_1 . The meaning of $+_D$, $-_D$, $++_D$, $--_D$ is dual w.r.t. $+_S$, $-_S$, $++_S$, $--_S$ respectively (by “dual” we mean that we invert satisfiability with deniability). The relations $+$, $-$, $++$, $--$ are such that each $G_2 \xrightarrow{r} G_1$ is a shorthand for the combination of the two corresponding relationships $G_2 \xrightarrow{r_S} G_1$ and $G_2 \xrightarrow{r_D} G_1$. (We call the first kind of relations *symmetric* and the latter two *asymmetric*.)

Let G_1, G_2, \dots denote goal labels. We introduce four distinct predicates over goals, $FS(G)$, DG and $PS(G)$, $PD(G)$, meaning respectively that there is (at least) *full* evidence that goal G is satisfied and that G is denied, and that there is at least *partial* evidence that G is satisfied and that G is denied.

To formalize the propagation of satisfiability and deniability evidence in a goal graph, we introduce in (Giorgini et al., 2002) a set of axioms stating: full satisfiability and deniability imply partial satisfiability and deniability, respectively; for an AND relation, full and partial satisfiability of the target node require respectively the full and partial satisfiability of all the source nodes; satisfiability (but not the full satisfiability) propagates through a “ $+_S$ ” relation. Thus, an AND relation propagates the minimum satisfiability value (and the maximum deniability one), while a “ $+_S$ ” relation propagates at most a partial satisfiability value. Dual axioms hold for the other relations.

Given a goal graph, we can perform two different kind of reasoning: *Top-Down* and *Bottom-Up*. In Top-Down reasoning, we concentrate on a set of root goals with a desired assignment (e.g., satisfy all of them), and we want to find an assignment to the leaf nodes consistent with the desired assignment. In other words, we want to find an initial assignment to the leaf nodes that can be propagate the desiderata assignment to the root nodes. In Bottom-Up reasoning, we concentrate on a set of leaf nodes with an initial assignment, and propagate these assignments upwards to find out their implications for root-level goals.

In (Giorgini et al., 2002) we have proposed sound and complete algorithms for qualitative and quantitative Top-Down reasoning with goal models. In particular, given a goal model and labels for some of the goals, our algorithms propagate these labels upwards. If the graph contains loops, propagation proceeds until a fixpoint is reached. We have also developed algorithms for qualitative and quantitative Bottom-Up reasoning.

6. Conclusions

We have presented an overview of the Tropos methodology. The basic assumption that distinguishes our work from others in Requirements Engineering is that actors and goals are used as fundamental concepts for mod-

eling and analysis during all phases of software development, not just early requirements. The distinguishing feature of Tropos compared to other agent-oriented software development methodologies is its emphasis on requirements analysis. Further information about the Tropos project can be found at <http://www.troposproject.org>.

The methodology has only been applied so far to several modest-size case studies, e.g., (Castro et al., 2002), with encouraging results. Anyway, it still lacks tools that support the transition between different phases. Another limitation of the methodology is that it has not been used for the development of full-fledged MAS.

Of course, much remains to be done to further refine and evaluate the proposed methodology. We are currently working on several open problems, such as the development of other formal analysis techniques for Tropos models, and the development of tools that support design activities during different phases of the methodology.

Acknowledgements

We thank all contributors to the Tropos project – in Trento, Toronto and elsewhere – for useful comments, discussions and feedback. This research has been partly supported by the Italian MIUR-FIRB Project, RBNE0195K5 - ASTRO and by the Natural Sciences and Engineering Research Council (NSERC) of Canada.

Chapter 6

THE MASE METHODOLOGY

Scott A. DeLoach

Abstract MaSE provides a detailed approach to the analysis and design of MAS. MaSE combines several established models into a comprehensive methodology and provides a set of transformation steps that shows how to derive new models from the existing models. Thus MaSE guides the developer in the analysis and design process. Future work on MaSE will focus on specializing it for use in adaptive multiagent and cooperative robotic systems based on an organizational theoretic approach. We are currently developing an organizational model that will provide the knowledge required for a team of software or hardware agents to automatically adapt to changes in their environment and to organize and re-organize to accomplish team goals. Much of the information needed in this organizational model – goals, roles, and agents – is already captured in MaSE. However, we will have to extend MaSE analysis to capture more detail on roles, including the capabilities required to play roles.

1. Introduction

This chapter provides an introduction to Multiagent Systems Engineering (MaSE), which is a full-lifecycle methodology for analyzing, designing, and developing heterogeneous MAS. To accomplish this, MaSE uses a number of graphically based models derived from standard UML models to describe the types of agents in a system and their interfaces to other agents, as well as an architecture-independent detailed definition of the internal agent design. The primary focus of MaSE is to guide a designer from an initial set of requirements through the analysis, design, and implementation of a working MAS.

MaSE views MAS as a further abstraction of the object-oriented paradigm where agents are specialized objects. Instead of simple objects, with methods that can be invoked by other objects, agents coordinate with each other via conversations and act proactively to accomplish individual and system-wide goals. Therefore, MaSE builds upon well-founded object-oriented techniques and applies them to the specification and design of MAS.

MaSE is also the basis for the *agentTool* development system. *agentTool* can be downloaded free from the *agentTool* Web page at <http://www.cis.ksu.edu/~sdeloach>. *agentTool* is a graphically based, fully interactive software engineering tool, which fully supports each step of MaSE analysis and design. *agentTool* also supports automatic verification of inter-agent communications, semi-automated design, and code generation for multiple MAS frameworks. MaSE and *agentTool* are both independent of any particular agent architecture, programming language, or communication framework.

2. Methodology

The MaSE methodology is a specialization of more traditional software engineering methodologies. The general operation of MaSE follows the phases and steps shown below and uses the associated models.

Phases	Models
1. Analysis Phase	
a. Capturing Goals	Goal Hierarchy
b. Applying Use Cases	Use Cases, Sequence Diagrams
c. Refining Roles	Concurrent Tasks, Role Model
2. Design Phase	
a. Creating Agent Classes	Agent Class Diagrams
b. Constructing Conversations	Conversation Diagrams
c. Assembling Agent Classes	Agent Architecture Diagrams
d. System Design	Deployment Diagrams

The MaSE Analysis phase consists of three steps: Capturing Goals, Applying Use Cases, and Refining Roles. The Design phase has four steps: Creating Agent Classes, Constructing Conversations, Assembling Agent Classes, and System Design. While presented sequentially, the methodology is, in practice, iterative. The intent is that the designer is free to move between steps and phases such that with each successive pass, additional detail is added and, eventually, a complete and consistent system design is produced.

One strength of MaSE is the ability to track changes during the whole process. Every object created during the analysis and design phases can be traced forward or backward through the different steps to other related objects. For instance, a goal derived in the Capturing Goals step can be traced to a specific role, task, and agent class. Likewise, an agent class can be traced back through tasks and roles to the system level goal it was designed to satisfy.

3. Analysis Phase

The MaSE Analysis phase produces a set of roles and tasks, which describe how a system satisfies its overall goals. Goals are an abstraction of the detailed

requirements and are achieved by roles. Typically, a system has an overall goal and a set of sub-goals that must be achieved to reach the system goal. Goals are used in MaSE because they capture *what* the system is trying to achieve and tend to be more stable over time than functions, processes, or information structures.

A *role* describes an entity that performs some function within the system. In MaSE, each role is responsible for achieving, or helping to achieve specific system goals or sub-goals. MaSE roles are analogous to roles played by actors in a play or by members of a typical company structure. The company (which corresponds to system) has roles such as “president,” “vice-president,” and “mail clerk” that have specific responsibilities, rights and relationships defined in order to meet the overall company goal.

The overall approach in the MaSE Analysis phase is straightforward: define system goals from a set of requirements and then define the roles necessary to meet those goals. To help in defining roles to meet specific goals, MaSE uses Use Cases and Sequence Diagrams. The individual steps of the Analysis phase of Capturing Goals, Applying Use Cases, and Refining Roles are presented next.

3.1 Capturing Goals

The first step in the MaSE Analysis phase is Capturing Goals, whose purpose is to transform an initial system specification into set of structured system goals. The *initial system context*, the starting point for MaSE analysis, is usually a software requirement specification with a well-defined set of requirements. These requirements tell the analyst the services that the system must provide and how the system should or should not behave based on inputs to the system and its current state. There are two sub-steps in Capturing Goals: identifying goals and structuring goals. First, goals must be identified from the initial system context. Next, the goals are analyzed and put into a hierarchical form. Each of these sub-steps is described in detail below.

Identifying Goals. The goal of the step named Identifying Goals is to capture the essence of an initial set of requirements. This process begins by extracting scenarios from the initial specification and describing the goal of that scenario.

Throughout this chapter, we will use the conference management system example, which has become fairly common in AOSE circles. The conference management system is a MAS supporting the management of various sized international conferences that require the coordination of several individuals and groups. We define the basic system requirements below.

- Authors should be able to submit their papers electronically to a conference paper database system. During the submission phase, authors should be notified of paper receipt and given a paper submission number.
- After the deadline for submissions has passed, the papers will be divided among the program committee (PC), who has to review the papers themselves or by contacting referees and asking them to review a number of the papers.
- Reviewers should be able to get papers directly from the central database and submit their reviews to a central collection point.
- After the reviews are complete, a decision on accepting or rejecting each paper must be made.
- After the decisions are made, authors are notified of the decisions and are asked to produce a final version of their paper if it was accepted.

The conference management system is an organization whose membership (authors, reviewers, decision makers, review collectors, etc.) may change during the process. In addition, since each agent is associated with a human, it is easy to imagine that these agents could be coerced into displaying opportunistic behaviors that would benefit their owner to the detriment to the overall system. Such behaviors could include reviewing ones own paper or inequitable allocation of work, etc.

An example of the goals derived from these requirements is shown below. Notice that all the details on how to perform system functions are not included as goals.

1. Collect papers
2. Distribute papers
3. Assign papers to PC members
4. Assign papers to reviewers
5. Submit reviews
6. Collect reviews
7. Select/reject papers
8. Inform authors

Goals embody the critical system requirements; therefore, an analyst should specify goals as abstractly as possible without losing the spirit of the requirement. This abstraction can be performed by removing detailed information when specifying goals. For example, to “Inform authors” is a goal, the details on how to actually inform them may change over time and are not.

Once the goals have been captured, they provide the foundation for the analysis model; all roles and tasks defined in later steps must support one of the goals. If, later in the analysis, the analyst discovers roles or tasks that do not support an existing system goal, either the roles and tasks are superfluous or a new goal has been discovered.

Structuring Goals. The final step in Capturing Goals is structuring the goals into a Goal Hierarchy Diagram, as shown in Figure 6.1. A Goal Hierarchy Diagram is a directed, acyclic graph where the nodes represent goals and

the arcs define a sub-goal relationship. A goal hierarchy is not necessarily a tree as a goal may be a sub-goal of more than one parent goal.

To develop the goal hierarchy, the analyst studies the goals for their importance and inter-relationships. Even though goals have been captured, they are of various importance, size, and level of detail. The Goal Hierarchy Diagram preserves such relationships, and divides goals into sub-goals that are easier to manage and understand.

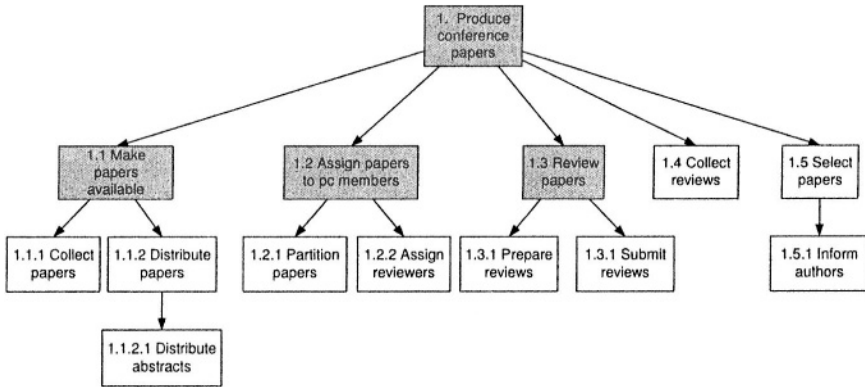


Figure 6.1. Example Goal Hierarchy Diagram

The first step in building is to identify the overall system goal, which is placed at the top of the Goal Hierarchy Diagram. However, it is often the case, as in our example above, that a single system goal cannot be directly extracted from the basic requirements. In this case, the highest-level goals are summarized to create an overall system, in our case “Produce conference papers.” Once a basic goal hierarchy is in place, goals may be decomposed into new sub-goals. Each sub-goal must support its parent goal in the hierarchy and defines *what* must done to accomplish the parent goal.

Although similar, Goal decomposition is not simply “functional decomposition.” Goals describe *what*, while functions describe *how*. Instead of a set of goals describing *what* the system will do, functional decomposition typically results in a set of steps prescribing *how* the system will do it. For example, functional steps for implementing the goal “Assign papers to PC members” might be to (i) group papers based on similar keywords; and (ii) select PC members whose expertise matches the paper groups. However, the appropriate sub-goals would be to: (i) “Partition papers”; and (ii) “Assign reviewers.” The fact that the papers are partitioned and PC members are assigned to papers are goals, *how* we divide the papers or on what basis we assign reviewers are immaterial at this point and will be decided on by the agents responsible for those goals. Goal decomposition continues until any further decomposition

would result in functions instead of a goals (i.e., the analyst prescribes *how* a goal should be accomplished).

There are four special types of goals in a Goal Hierarchy Diagram. These are: summary, partitioned, combined, and non-functional. Goals can have attributes of more than one special goal type; however, they do not necessarily have to be one of these types at all.

A *summary goal* is derived from a set of existing “peer” goals to provide a common parent goal. This often happens at the highest levels of the hierarchy as was the case in the overall system goal in our example.

Some goals do not functionally support the overall system goal, but are critical to system operation. These *non-functional goals* are often derived from non-functional requirements such as reliability or response times. For example, if a system must be able to find resources dynamically, a goal to facilitate locating dynamic resources may be required. In this case, another “branch” of the Goal Hierarchy Diagram can be created and placed under an overall system level goal.

There are often a number of sub-goals in a hierarchy that are identical or very similar that can be grouped into a *combined goal*. This often happens when the same basic goal is a sub-goal of two different goals. In this case, the combined goal becomes a sub-goal of both the goals.

A *partitioned goal* is a goal with a set of sub-goals that, when taken collectively, effectively meet that goal. While this is always true of summary goals, it may be true of any goals with a set of sub-goals. By defining a goal as “partitioned,” it frees the analyst from specifically accounting for it in the rest of the analysis process. Partitioned goals are annotated in a Goal Hierarchy Diagram using a gray goal box instead of a clear box (e.g., goals 1, 1.1, and 1.2 in Figure 6.1).

At the conclusion of Capturing Goals, system goals have been captured and structured into a Goal Hierarchy Diagram. The analyst can now move to the second Analysis step, Applying Use Cases, where the initial look at roles and communication paths takes place.

3.2 Applying Use Cases

The Applying Uses Cases step is crucial in translating goals into roles and associated tasks. Use cases are drawn from the system requirements and describe sequences of events that define desired system behavior; they are examples of how the system should behave. To help determine the actual communications in a MAS, the use cases are converted into Sequence Diagrams. MaSE Sequence Diagrams are similar to standard UML sequence diagrams except that they are used to depict sequences of events between *roles* and to define the communications between the agents that will be playing those roles. The roles

identified here form the initial set of roles used in the next step while the events are also used later to define tasks and conversations.

The first step in Applying Use Cases is to extract Use Cases from the initial system context, which should include both positive and negative Use Cases. A *positive Use Case* describes what should happen during normal system operation. However, a *negative Use Case* defines a breakdown or error. While Use Cases cannot be used to capture every possible requirement, they are an aid in deriving communication paths and roles. Cross checking the final analysis against the set of derived goals and Use Cases provides a redundant method for deriving system behavior.

3.3 Refining Roles

The purpose of the Refining Roles step is to transform the Goal Hierarchy Diagram and Sequence Diagrams into roles and their associated tasks, which are forms more suitable for designing MAS. Roles form the foundation for agent classes and correspond to system goals during the Design phase. It is our contention that system goals will be satisfied if every goal is associated with a role and every role is played by an agent class.

The general case transformation of goals to roles is one-to-one, with each goal mapping to a role. However, there are situations where it is useful to have a single role be responsible for multiple goals, including convenience or efficiency. One mapping of the goals from our previous example to a set of roles is shown below.

PaperDB	(1.1.1, 1.1.2, 1.1.2.1)
Partitioner	(1.2.1)
Assigner	(1.2.2)
Reviewer	(1.3.1)
Collector	(1.4)
DecisionMaker	(1.5, 1.5.1)

Due to the simplicity of our example, we mapped goals to individual roles with a two exceptions. Goals, 1, 1.1, 1.2, and 1.3 were not mapped to roles since they were partitioned. However, the PaperDB role was assigned all the goals associated with goal 1.1, namely 1.1.1, 1.1.2, and 1.1.2.1. In addition, the DecisionMaker role was assigned both 1.5 and 1.5.1, which are closely related.

Related goals can often be combined into a single role. For example, the “collect papers,” “distribute papers,” and “distribute abstracts” goals are combined into the single PaperDB role since they are closely related and require the same type of access techniques. While combining goals makes the role more complex, it may simplify the overall design.

In general, interfacing with external or internal resources requires a separate role to act as an interface to the rest of the system. We generally consider

a human user as an external resource. In MaSE we do not explicitly model human-computer interaction; we create a specific role to encapsulate the user interface. In this way, we can define the ways in which a user can interface with the system without defining the user interface itself. Other resources such as databases, files or legacy systems may also require their own interface role. In our example, the Author role does not satisfy any system goals as it is an interface to the user; however, without it, the system is not needed.

Role definitions are captured in a MaSE Role Model as shown in Figure 6.2, which includes information on interactions between role tasks and is more complex than traditional role models, as described in (Kendall, 1998). Roles are denoted by rectangles, while a role's tasks are denoted by ovals attached to the role. Lines between tasks denote communications protocols with the arrow pointing from the initiator to the respondent. Solid lines indicate external communications while dashed lines denote communication between tasks in the same role instance.

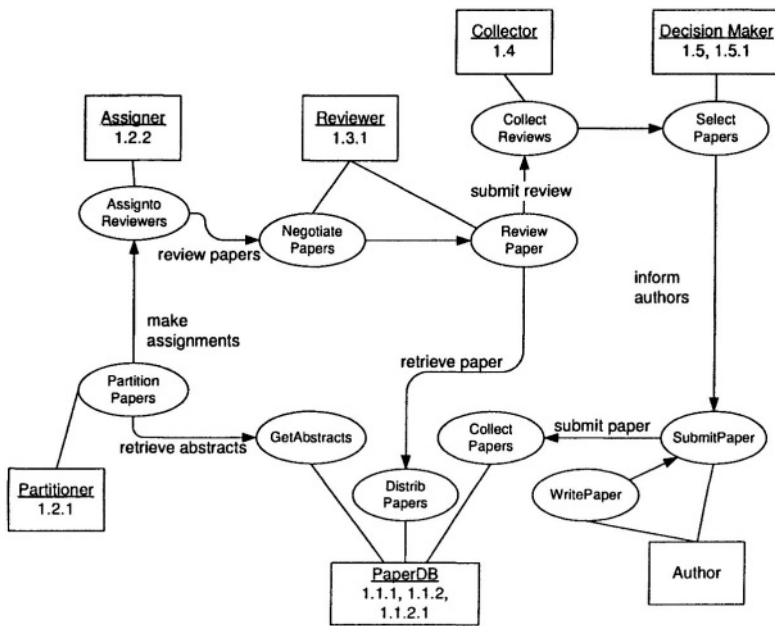


Figure 6.2. MaSE Role Model

The tasks are generally derived from the goals for which a task is responsible. For instance, the PaperDB role is responsible for attaining goals 1.1.1, 1.1.2, and 1.1.2.1. Therefore, to accomplish this goal, the role must be able to collect papers and distribute them and their abstracts. Therefore, we created three interrelated tasks: Collect Papers, Distrib Papers, and GetAbstracts.

While we could have specified all three goals in a single task, partitioning them in this way is modular and effectively encapsulates the actual approach used.

Roles should not share or duplicate tasks. Sharing of tasks is a sign of improper role decomposition. Shared tasks should be placed in a separate role, which can be combined into various agent classes in the Design phase.

Concurrent Task Model. After roles are created and tasks identified, the developer captures the role’s behavior by defining the details of the individual tasks. A role may consist of multiple tasks that, when taken together, define the required behavior of that role. Each task executes in its own thread of control, but may communicate with each other. Concurrent tasks are defined in Concurrent Task Models (see Figure 6.3) and are specified as finite state automata, which consist of states and transitions. *States* encompass the processing that goes on internal to the agent while *transitions* allow communication between agents or between tasks.

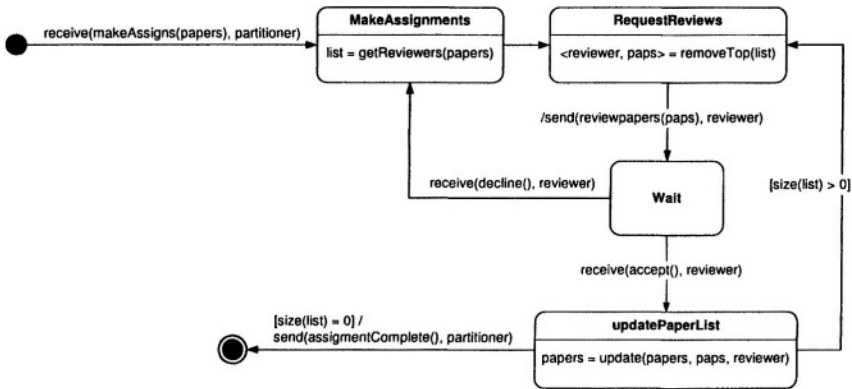


Figure 6.3. Concurrent Task Diagram

A transition consists of a source state, destination state, trigger, guard condition, and transmissions and uses the syntax *trigger* [*guard*] ^ *transmission(s)*. Multiple transmissions may be separated with a semicolon (;), however, no ordering is implied. Generally, events on triggers or transmissions are to be associated with a task within the same role, thus allowing internal task coordination. However, two special events, send and receive, are used to indicate messages sent between agents. The *send* event (denoted *send(message, agent)*) is used to send a message to another agent while the *receive* event (denoted as *receive(message, agent)*) signifies the receipt of a message. The *message* is defined as a performative, which describes the intent of the message, along with a set of parameters that are the content of the message (i.e., *performa-*

tive(p1 ... pn) where *p1 ... pn* denotes *n* parameters). It is also possible to send a message to a group of agents via multicasting using a *<group-name>* versus a single agent name.

States may contain *activities* that represent internal reasoning, reading a percept from sensors, or performing actions via actuators. Multiple activities may be included in a single state and are performed in an uninterruptable sequence. Once in a state, the task remains there until the activity sequence is complete. The variables used in activity and events definitions are visible within the task, but not outside of the task or within activities. All messages sent between roles and events sent between tasks are queued to ensure that all messages are received even if the agent or task is not in the appropriate state to handle the message or event immediately.

Concurrent tasks have predefined activities to deal with mobility and time. The *move* activity specifies that the agent is to move to a new address and returns a Boolean value (*Boolean = move(location)*), which states whether the move actually occurred. The agent can reason over this value and deal with it accordingly.

To reason about time, the Concurrent Task Model provides a built in timer activity. An agent can define a timer using *t = setTimer(time)*, the *setTimer* activity. The *setTimer* activity takes a time as input and returns a timer that will timeout in exactly the time specified. The timer that can then be tested via the *timeout* activity, which returns a Boolean value, to see if it has “timed out” (*Boolean = timeout(t)*).

Once a transition is enabled, it is executed instantaneously. If multiple transitions are enabled, the following priority scheme is used.

- 1 Transitions whose triggers are internal events.
- 2 Transitions whose transmissions are internal events.
- 3 Transitions whose trigger receives a message from another role.
- 4 Transitions whose transmissions are a message to another role.
- 5 Transitions with valid guard conditions only.

Figure 6.3 shows the *Assign to Reviewers* task for the Assigner role. The task is initiated upon receipt of a *makeAssigns* message from a Partitioner agent, which includes a list of papers to be assigned. After the message is received, the task goes to the *MakeAssignments* state where it computes a list of reviewers for the papers (a process that is as yet undefined). Once this list is defined, the task transitions to the *RequestReviews* state where the top reviewer/papers tuple is taken off the list. A *reviewPapers* message is then sent to the reviewer effectively requesting that the agent provide a review for the

associated papers, which is denoted by the *paps* parameter. The task remains in the Wait state until a reply from the reviewer is received. If the reviewer declines (via a decline message), the task returns to the MakeAssignment state where it computes a new list of reviewers for the remaining papers. If the reviewer accepts the request via an accept message, the task transitions to the updatePaperList state where the list of papers is updated by adding the name of the reviewer to the papers that they will be reviewing. If the list is not empty, the task returns to the RequestReviews state to make a request of the next reviewer on the list. If the size of the reviewers list is empty, the task ends by sending an assignmentComplete message to the Partitioner agent.

3.4 Analysis Phase Summary

Once the concurrent tasks of each role are defined, the Analysis phase is complete. The MaSE Analysis phase is summarized as follows:

- 1 Identify goals and structure them into a Goal Hierarchy Diagram.
- 2 Identify Use Cases and create Sequence Diagrams to help identify roles and communications paths.
- 3 Transform goals into a set of roles.
 - (a) Create a Role Model to capture roles and their tasks.
 - (b) Define role behavior using Concurrent Task Models for each task.

4. Design Phase

There are four steps to the designing a system with MaSE. The first step is Creating Agent Classes, in which the designer assigns roles to specific agent types. In the second step, Constructing Conversations, the conversations between agent classes are defined while in the third step, Assembling Agents Classes, the internal architecture and reasoning processes of the agent classes are designed. Finally, in the last step, System Design, the designer defines the number and location of agents in the deployed system.

4.1 Creating Agent Classes

In the Creating Agent Classes step, agent classes are created from the roles defined in the Analysis phase. This phase produces an Agent Class Diagram, which depicts the overall agent system organization consisting of agent classes and the conversations between them. An *agent class* is a template for a type of agent in the system and are defined in terms of the roles they will play and the conversations in which they may participate. If roles are the foundation of MAS design, then agent classes are the bricks used to implement MAS.

These two different abstractions manipulate two distinct system dimensions. Roles allow us to allocate system goals while agent classes allow us to consider communications and other resource usage.

The first step is to assign roles to each agent class. If assigned multiple roles, agent classes may play them concurrently or sequentially. To ensure that system goals are accounted for, each role must be assigned to at least one agent class. The analyst can easily change the organization and allocation of roles among agent classes during design, since roles can be manipulated modularly. This allows consideration of various design issues, which are based on standard software engineering concepts such as functional, communicational, procedural, or temporal cohesion.

During this step, we also identify the conversations in which different agent classes must participate. An agent's conversations are derived from the external communications of the agent's assigned roles. For instance, if roles A and B communicate with each other, then, if agent 1 plays role A and agent 2 plays role B, then there must be a conversation between agent 1 and agent 2.

The agent classes and conversations are documented via Agent Class Diagrams, which are similar to object-oriented class diagrams with two main differences. First, agent classes are defined by the roles they play, not by attributes and methods. Second, all relationships between agent classes are captured as conversations. A sample Agent Class Diagram is shown in Figure 6.4. The boxes in Figure 6.4 denote agent classes and contain the class name and the set of roles each agent plays. Lines with arrows identify conversations and point from the conversation initiator to the responder. In this design, the PC Chair agent plays the Partitioner, Collector, and Decision Maker roles while the PC Member agent plays both the Assigner and Reviewer roles. Outside of Authors, the only other agent is the DB agent, which provides an interface to the database containing papers, abstracts, and author information.

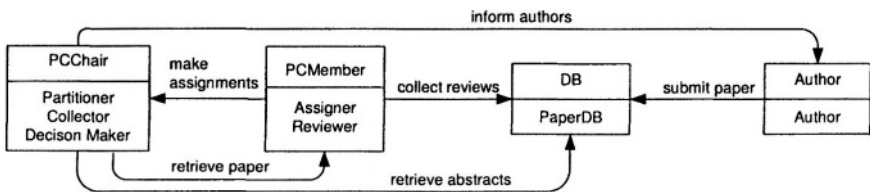


Figure 6.4. Agent Class Diagram

The Agent Class Diagram is the first design object in MaSE that depicts the entire MAS in its final form. If we have carefully followed MaSE to this point, the system represented by the Agent Class Diagram will support the goals and Use Cases identified in the Analysis phase. Of particular importance at this

point is the system organization – the way that the agent classes are connected with conversations.

4.2 Constructing Conversations

Constructing Conversations is the next MaSE Design phase step. So far, the designer has only identified conversations; the goal of this step is to define the details of those conversations based on the internal details of concurrent tasks.

A *conversation* defines a coordination protocol between two agents and is documented using two Communication Class Diagrams, one each for the initiator and responder. A Communication Class Diagram, as shown in Figure 6.5, is similar to a Concurrent Task Model and defines the conversation states of the two participant agent classes. The initiator begins the conversation by sending the first message. When the other agent receives the message, it compares it to its active conversations. If it finds a match, the agent transitions the appropriate conversation to a new state and performs any required actions or activities from either the transition or the new state. Otherwise, the agent assumes the message is a new conversation request and compares it to the conversations it can participate in with the sending agent. If the agent finds a match, it begins a new conversation.

As stated above, communication class diagrams use states and transitions to define the inter-agent communication. Transitions use the following syntax: *rec-mess(args1) [cond] / action ^ trans-mess(args2)*. This states that if the message *rec-mess* is received with the arguments *args1* and the condition *cond* holds, then the method *action* is called and the message *trans-mess* is sent with arguments *args2*.

The transition from the start state in Figure 6.5 (left) indicates that it is the initiator half of a conversation, since it transmits a message. The conversation describes how the PCChair agent (the conversation initiator) sends a message to the Author agent notifying it of the acceptance. At this point, the PCChair enters a wait state. If the Author can still attend the conference, it sends an accept message (Figure 6.5 right) and the conversation is completed. If the Author cannot attend the conference, it returns a decline message. After receiving a decline message, the PCChair performs the *updatePapers* activity to update its list of attendees.

As discussed above, the designer establishes an agent's set of conversations by the roles it has been assigned. In the same way, the conversation design is derived from the concurrent tasks associated with those roles. Since a concurrent task integrates inter- and intra-role interactions, it provides the information required to define conversations. Each task that defines external communication creates one or more conversations. If all task communication is with a single role, or set of roles that have all been mapped to a single agent class, the

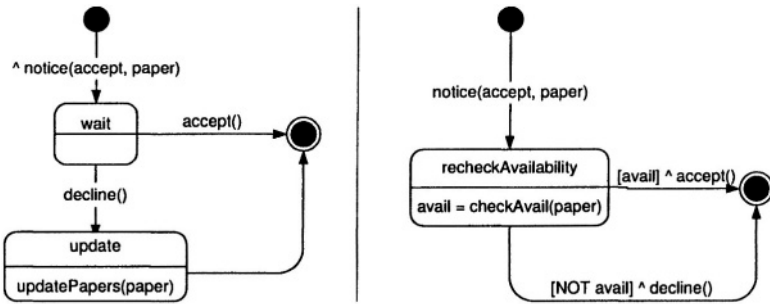


Figure 6.5. Inform authors conversation initiator and responder

task can be mapped directly to a single conversation. More generally, however, concurrent tasks spawn multiple conversations.

Once the information from Concurrent Task Models has been integrated into conversations, the designer must ensure that other factors, such as robustness and fault tolerance, are taken into account. For instance, if a particular agent sends a message to another agent requesting an action be performed, the conversation should be able to handle the other agent's refusal or inability to complete the request.

4.3 Assembling Agents

Agent class internals are designed during the step Assembling Agents, that includes two sub-steps: defining the architecture of agents and defining the architecture's components. Designers have the choice of either designing their own architecture or using predefined architectures such as BDI. Likewise, a designer may use predefined components or develop them from scratch. Components consist of a set of attributes, methods, and possibly a sub-architecture.

An example of an Agent Architecture Diagram is shown in Figure 6.6. Architectural components (denoted by boxes) are connected to either inner- or outer-agent connectors. *Inner-agent connectors* (thin arrows) define visibility between components while *outer-agent connectors* (thick dashed arrows) define external connections to resources such as agents, sensors and effectors, databases, and data stores. Internal component behavior may be represented by formal operation definitions or state-diagrams. The architecture and internal definition of the components must be consistent with the conversations defined in the previous step. At a minimum, this requires that each action or activity defined in a Communication Class Diagram be defined as an operation in one of the internal components. The internal component state diagrams and operations can also be used to initiate and coordinate various conversations.

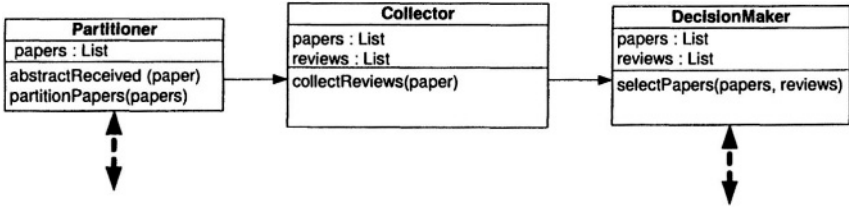


Figure 6.6. PCChair Agent Architecture

The PCChair agent architecture is shown in Figure 6.6. The PCChair agent has three components, which basically implement a pipeline architecture. The Partitioner component receives abstracts and uses the partitionPapers method to break the list into sets based on content. The Partitioner then calls the collectReviews method of the Collector component, which waits and collects all the reviews from the reviewer. Once all papers have been reviewed, the Collector component calls the selectPapers method of the DecisionMaker component, who selects the best papers and notifies the authors.

4.4 System Design

System Design is the final step of the MaSE methodology and uses Deployment Diagrams to show the numbers, types, and locations of agent instances in a system. System design is actually the simplest step of MaSE, as most of the work was done in previous steps. Figure 6.7 shows a Deployment Diagram for the conference management system. The three-dimensional boxes represent agents while the connecting lines represent actual conversations between agents. The agents are identified by their class name in the form of *instance-name : class*. Dashed boxes define physical computational platforms.

A designer should define the system deployment before implementation since agents typically require Deployment Diagram information, such as a hostname or address, for communications. Deployment Diagrams also offer an opportunity for the designer to tune the system to its environment to maximize available processing power and network bandwidth. In some cases, the designer may specify a particular number of agents in the system or the specific computers on which certain agents must reside. The designer should also consider the communication and processing requirements when assigning agents to computers. To reduce communications overhead, a designer may choose to deploy agents on the same machine. However, too many agents on a single machine destroys the advantages of distribution gained by using the multiagent paradigm. Another strength of MaSE is that a designer can make these mod-

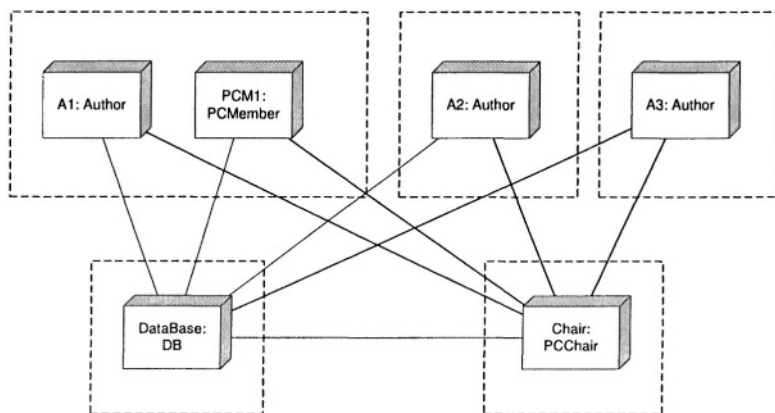


Figure 6.7. Deployment Diagram

ifications after designing the system organization, thus generating a variety of system configurations.

4.5 Design Phase Summary

Once the Deployment Diagrams are finished, the Design phase is complete. The MaSE Design Phase can be summarized as follows:

- 1 Assign roles to agent classes and identify conversations.
- 2 Construct conversations, adding messages/states for robustness.
- 3 Define internal agent architectures.
- 4 Define the final system structure using Deployment Diagrams.

5. agentTool

The agentTool system (DeLoach and Wood, 2001) has been developed to support and enforce MaSE. Currently agentTool implements all seven steps of MaSE as well as automated design support. The agentTool user interface is shown in Figure 6.8. The menus across the top allow access to several system functions, including analysis to design transformations (Sparkman et al., 2001), conversation verification (Lacey et al., 2000), and code generation. The buttons on the left add specific items to the diagrams while a text window displays system messages. The different MaSE diagrams are accessed via the tabbed panels across the top of the main window. When a MaSE diagram is selected, the designer can manipulate it graphically in the window. Each panel

has different types of objects and text that can be placed on them. Selecting an object in the window enables other related diagrams to become accessible.

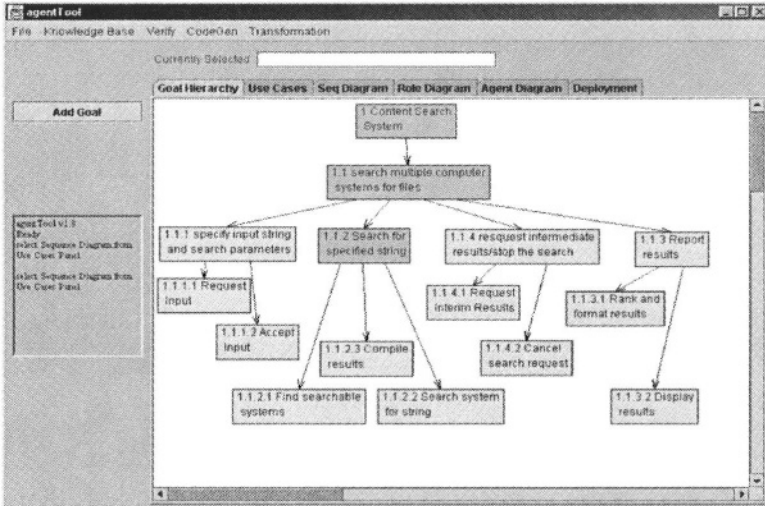


Figure 6.8. agentTool

While the designer may use existing architectures or design a new one from scratch, agentTool also provides the ability to semi-automatically derive the agent architecture directly from the roles and tasks defined in the analysis phase. This approach has the advantage of providing a direct mapping from analysis to design. Each task from each role played by an agent defines a component in the agent class. The concurrent task itself is transformed into a combination of the component's internal state diagram and a set of conversations. Activities identified in the concurrent task become methods of the component.

The transformation is actually a sequence of transformations that incrementally change roles and tasks into agent classes, components, and conversations. Before beginning the analysis-to-design transformation process, the Role Model and its set of concurrent tasks, and the assignment of roles to agent classes must exist. During the first stage of the transformation process, agentTool derives agent components from their assigned roles and assigns external events to specific protocols. In the second stage, agentTool annotates the component state diagrams to determine where conversations start and end. During the last stage, agentTool extracts the annotated states and transitions and uses them to create new conversations, replacing them in the component state diagram with actions initiating the conversation.

A second set of transformations that is currently implemented in agentTool consists of transformations to add functionality required for mobility. In the

analysis phase, mobility is specified using a *move* activity in the state of a concurrent task diagram. This *move* activity is copied directly into the associated component state diagram during the initial set of analysis-to-design transformation described above. During the mobility transformation, the existing design is modified to coordinate the mobility requirements between all components in the agent design. In the derived mobility design, the Agent-Component is responsible for coordinating the entire move and working with the external agent platform to save its current state and actually carry out the move.

The agentTool system also provides automatic verification of conversations. The verification process begins with the fully automated translation of system conversations into the Promela modeling language. Then, the Promela model is automatically analyzed using the Spin verification tool to detect errors such as deadlock, non-progress loops, syntax errors, unused messages, and unused states (Holzmann, 1997). Feedback is provided to the designer automatically via text messages and graphical highlighting of error conditions.

6. Applications

MaSE has been successfully applied in many graduate-level projects as well as several research projects. The Multiagent Distributed Goal Satisfaction project used MaSE to design the collaborative agent framework to integrate different constraint satisfaction and planning systems. The Agent-Based Mixed-Initiative Collaboration project also used MaSE to design a MAS focused on distributed human and machine planning. MaSE has been used successfully to design an agent-based heterogeneous database system as well as a multiagent approach to a biologically based computer virus immune system. More recently, we applied MaSE to a team of autonomous, heterogeneous search and rescue robots (DeLoach et al., 2003). The MaSE approach and models worked very well. The concurrent tasks mapped nicely to the typical behaviors in robot architectures. MaSE also provided the high-level, top-down approach missing in many cooperative robot applications.

7. Comparison with other Methodologies

There have been several methodologies proposed for developing MAS (see chapter 7). However, we only compare MaSE against the two other methodologies presented in this book: Gaia (see chapter 4) and Tropos (see chapter 5).

The Gaia method, as presented in chapter 4, is one of the best-known approaches to building MAS and has many similarities with MaSE. As in MaSE, Gaia uses roles as building blocks. In general, both the analysis phases of MaSE and Gaia capture much of the same type of information, although in different types of models. The major difference is in the level of support for

detailed agent design provided by Gaia. Gaia produces a high-level design and assumes the details will be developed using traditional techniques whereas MaSE provides models and guidance on creating the detailed design.

Tropos, which was presented in chapter 5, take a significantly different approach than MaSE. Tropos focuses on early requirements definition, which is not stressed with MaSE. Tropos uses Yu's *i** framework (Yu, 2001), which provides a nice front end to Tropos. In fact, the Tropos early requirements approach could be used with MaSE as the goal model of each methodology are essentially the same.

Chapter 7

A COMPARATIVE EVALUATION OF AGENT-ORIENTED METHODOLOGIES

Arnon Sturm and Onn Shehory

Abstract Multiple agent-oriented methodologies were introduced in recent years, however no systematic evaluation of these was offered. In this work we perform a comparative evaluation of three well-known agent-oriented methodologies: Gaia, Tropos, and MaSE. To perform this evaluation we use an existing framework that focuses on four major facets of a methodology: concepts and properties, notations and modeling techniques, development process, and pragmatics. Analyzing the results of our evaluation, we recognize several facets that need further improvements within the existing agent-oriented methodologies. Our study does not attempt to state what the right methodology is. Rather, it examines existing agent-oriented methodologies.

1. Introduction

During the last decade, many methodologies for developing agent-based systems have been developed. A methodology is the set of guidelines for covering the whole lifecycle of system development both technically and managerially. A methodology, according to (Graham et al., 1997), should provide the following: a full lifecycle process; a comprehensive set of concepts and models; a full set of techniques (rules, guidelines, heuristics); a fully delineated set of deliverables; a modeling language; a set of metrics; quality assurance; coding (and other) standards; reuse advice; and guidelines for project management. The relationships between these components are shown in Figure 7.1. In that figure, we use the UML notations to depict the relationships between the components. As depicted in the figure, a methodology consists of a set of techniques, a modeling language and a lifecycle process. The set of techniques consists of metrics, quality assurance (QA) activities, a set of standards and tools. The modeling language comprises notations and a meta model. The lifecycle process consists of project management, a number of roles (e.g., an analyst or a designer), a number of procedures (e.g., how to move between

development stages), and a number of deliverables (e.g., a design document, source code). In addition, Figure 7.1 shows that the tools should be based on the meta model of the modeling technique and should represent the modeling technique's notations. The deliverables should use the modeling technique.

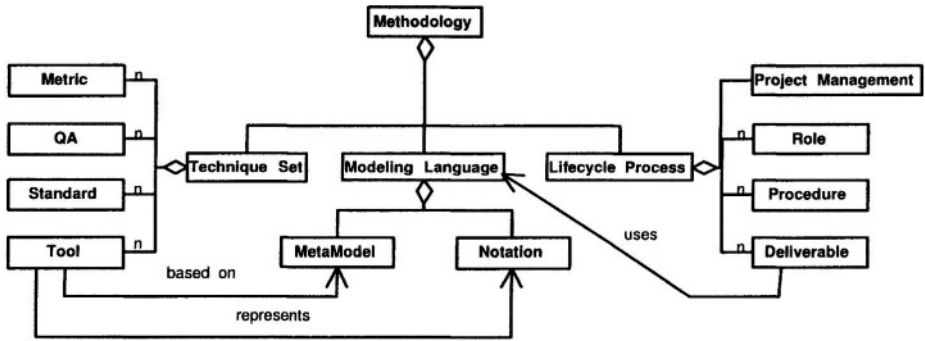


Figure 7.1. The components of a methodology and the relationships among them

At present, more than two dozens agent-oriented methodologies exist. The multiplicity and variety of methodologies result in the following problems: (i) industrial problem: selecting a methodology for developing an agent-based system/application becomes a non-trivial task, in particular for industrial developers which hold specific requirements and constraints; (ii) standards problem: multiple different methodologies are counter-productive for arriving at a standard. With no standard available, potential industrial adopters of agent technology refrain from using it; and (iii) research problems: excessive efforts are spent on developing agent-oriented methodologies, in times producing overlapping results. Additionally, as a result of allocating resources to multiple methodologies, no methodology is allocated sufficient research resources to enable addressing all facets and providing a full-fledged agent-based methodology.

A few evaluations of agent-oriented methodologies have been suggested. In (Yu and Cysneiros, 2002), the authors set a list of questions that a methodology should address. However, neither evaluation nor a comparison has been performed using that set. Another study (Cernuzzi and Rossi, 2002) suggests a framework for evaluating agent-oriented methodologies. That framework uses a set of evaluation criteria to examine methodologies' expressiveness, however it does not examine other properties encompassed within the methodology definition. In (Kumar, 2002), the author performs an evaluation of five agent-oriented methodologies, however, he refers only to some supported concepts such as organization design and cooperation and not to the broad set of attributes that constitute a complete methodology. In (Shehory and Sturm,

2001), the authors perform an evaluation of the modeling part within a methodology, however other parts are not evaluated. In (Dam and Winikoff, 2003), three methodologies were compared: MaSE (see chapter 6), Prometheus (see chapter 11) and Tropos (see chapter 5). The comparison was performed by gathering feedback regarding the properties of the methodologies from students that used them, and from the methodologies' developers. The gathered feedback included several inconsistent answers. This results in difficulty in analyzing methodology properties.

Other studies that deal with evaluating agent-oriented methodologies compared two or three methodologies, yet mainly with respect to the expressiveness of the methodologies and their supported concepts, and not with respect to other software engineering criteria.

In this chapter we evaluate three well-known methodologies: Gaia (see chapter 4), Tropos (see chapter 5), and MaSE (see chapter 6). Unlike previous research on the evaluation of agent methodologies, our evaluation examines multiple dimensions, possibly referring to all of the major facets relevant to methodology evaluation. We perform this evaluation relying on the evaluation framework suggest by (Sturm and Shehory, 2003). We present this evaluation framework in section 2. We then perform evaluations of the methodologies in sections 3 to 5. This series of evaluations is concluded in section 6.

2. The Evaluation Framework

The evaluation framework used in this chapter is based on a feature analysis technique. That is, the features of each of the examined methodologies are evaluated. The evaluation is performed based on information regarding the examined methodologies available in publications. The framework's four facets are: concepts and properties, notations and modeling techniques, development process, and pragmatics. These facets, and the metric used in conjunction with them, are introduced below.

2.1 Concepts and Properties

A concept is an abstraction or a notion inferred or derived from specific instances within a problem domain. A property is a special capability or a characteristic. This section deals with the question whether a methodology addresses the basic notions (concepts and properties) of agents and MAS. The following are the concepts according to which an agent-oriented methodology should be evaluated:

- 1 Autonomy: is the ability of an agent to operate without supervision;
- 2 Reactiveness: is the ability of an agent to respond in a timely manner to changes in the environment;

- 3 Proactiveness: is the ability of an agent to pursue new goals; and
- 4 Sociality: is the ability of an agent to interact with other agents by sending and receiving messages, routing these messages, and understanding them.

In the following we present the building blocks that encompass the basic components of MAS. These building blocks are based on (Sturm and Shehory, 2003).

- 1 Agent: is a computer program that can accept tasks, can figure out which actions to execute in order to perform these tasks and can actually execute these actions without supervision. It is capable of performing a set of tasks and providing a set of services.
- 2 Belief: is a fact that is believed to be true about the world.
- 3 Desire: is a fact of which the current value is false and the agent (that owns the desire) would prefer that it be true. Desires within an agent may be contradictory. A widely used specialization of a desire is a goal. The set of goals within an agent should be consistent.
- 4 Intention: is a fact that represents the way of realizing a desire. Sometimes referred to as a plan.
- 5 Message: is a means of exchanging facts or objects between entities.
- 6 Norm: is a guideline that characterizes a society. An agent that wishes to be a member of the society is required to follow all of the norms within. A norm can be referred to as a rule.
- 7 Organization: is a group of agents working together to achieve a common purpose. An organization consists of roles that characterize the agents, which are members of the organization.
- 8 Protocol: is an ordered set of messages that together define the admissible patterns of a particular type of interaction between entities.
- 9 Role: is an abstract representation of an agent's function, service, or identification within a group.
- 10 Society: is a collection of agents and organizations that collaborate to promote their individual goals.
- 11 Task: is a piece of work that can be assigned to an agent or performed by it. It may be a function to be performed and may have time constraints.

2.2 Notations and Modeling Techniques

Notations are a technical system of symbols used to represent elements within a system. A modeling technique is a set of models that depict a system at different levels of abstraction and different system's facets (e.g., structural and behavioral facets). This section deals with the properties to which methodology's notations and modeling techniques should adhere. The list of these properties is adopted from (Shehory and Sturm, 2001).

- 1 Accessibility: is an attribute that refers to the ease, or the simplicity, of understanding and using a method. It enhances both experts and novices capabilities of using a new concept.
- 2 Analyzability: is a capability to check the internal consistency or implications of models, or to identify aspects that seem to be unclear, such as the interrelations among seemingly unrelated operations. This capability is usually supported by automatic tools.
- 3 Complexity management (abstraction): is an ability to deal with various levels of abstraction (i.e., various levels of detail). Sometimes, high-level requirements are needed, while in other situations, more detail is required. For example, examining the top level design of a MAS, one would like to understand which agents are within the system, but not necessarily what their attributes and characterizations are. However, when concentrating on a specific task of an agent, the details are much more important than the system architecture.
- 4 Executability (and testability): is a capability of performing a simulation or generating a prototype of at least some aspects of a specification. These would demonstrate possible behaviors of the system being modeled, and help developers determine whether the intended requirements have been expressed.
- 5 Expressiveness (and applicability to multiple domains): is a capability of presenting system concepts that refers to:
 - The structure of the system;
 - The knowledge encapsulated within the system;
 - The system's ontology;
 - The data flow within the system;
 - The control flow within the system;
 - The concurrent activities within the system (and the agents);
 - The resource constraints within the system (e.g., time, CPU and memory);

- The system's physical architecture;
 - The agents' mobility;
 - The interaction of the system with external systems; and
 - The user interface specification.
- 6 Modularity (incrementality): is the ability to specify a system in an iterative incremental manner. That is, when new requirements are added it should not affect the existing specifications, but may use them.
- 7 Preciseness: is an attribute of disambiguity. It allows users to avoid misinterpretation of the existing models.

2.3 Development Process

A development process is a series of actions that, when performed, result in a working computerized system. This section deals with the process development facet of a methodology. This facet is evaluated by examining the following:

- 1 Development context: specifies whether a methodology can be used in creating new software, reengineering or reverse engineering existing software, prototyping, or designing for or with reuse components.
- 2 Lifecycle coverage: specifies what elements of software development are dealt with within the methodology. Each methodology may have elements that are useful in several stages of the development lifecycle. Here, the lifecycle stages are defined as follows: requirements' gathering, analysis, design, implementation, and testing.

Having the development stages defined is not sufficient to render a methodology usable. A methodology should further elaborate the activities within the development lifecycle. Providing a detailed description of the activities included in the development lifecycle would enhance the appropriate use of a methodology and increase its acceptability as a well-formed engineering approach. To verify that a methodology provides detailed activity descriptions, we need to examine the details of the development process. This verification can be performed by answering the following questions regarding an evaluated methodology:

- 1 What are the activities within each stage of a methodology? For example, an activity can be the identification of a role, a task, etc. The methodology may consist of heuristics or guidelines helping the developer to achieve his/her system development goals.

- 2 Does the process provide for verification? This question checks whether a methodology has rules for verifying adherence of its deliverables to the requirements.
- 3 Does the process provide for validation? This question checks whether a methodology has rules for validating that the deliverables of one stage are consistent with its preceding stage.
- 4 Are quality assurance guidelines supplied?
- 5 Are there guidelines for project management?

2.4 Pragmatics

Pragmatics refers to dealing with practical aspects of using a methodology. This section deals with pragmatics of adopting the methodology for a project or within an organization. In particular, the framework suggests examining the following:

- 1 Resources: are the (publicly available) publications describing in detail the methodology (e.g., textbooks and papers), users' groups, training and consulting services offered by third parties and automated tools (CASE tools) available in support of the methodology (e.g., graphical editors, code generators, and checkers).
- 2 Required expertise: is the required background of those learning the methodology. A distinguishing characteristic of many methodologies is the level of mathematical sophistication required to fully exploit the methodology. A criterion within the required expertise may check the required knowledge in some discipline.
- 3 Language (paradigm and architecture) suitability: is the level to which the methodology is coupled with a particular implementation language (e.g., object oriented programming language) or a specific architecture (e.g., BDI).
- 4 Domain applicability: indicates the level of suitability of a methodology to a variety of domains (e.g., information systems, real-time systems).
- 5 Scalability: is the ability of the methodology to be adjusted to handle various application sizes. For example, can it provide a lightweight version for simple problems.

2.5 Metric

To enable ranking of the properties examined in the evaluation process, the framework proposes a scale of 1 to 7 with the following interpretations:

- 1 Indicates that the methodology does not address the property.
- 2 Indicates that the methodology refers to the property but no details are provided.
- 3 Indicates that the methodology addresses the property to a limited extent. That is, many issues that are related to the specific property are not addressed.
- 4 Indicates that the methodology addresses the property, yet some major issues are lacking.
- 5 Indicates that the methodology addresses the property, however, it lacks one or two major issues related to the specific property.
- 6 Indicates that the methodology addresses the property with minor deficiencies.
- 7 Indicates that the methodology fully addresses the property.

Thus far, we have described the evaluation framework, its evaluation criteria, and its metric. Using these, we proceed with evaluating the Gaia, Tropos and MaSE methodologies.

3. Evaluating Gaia

In this section we evaluate Gaia according to the framework presented in section 7.2. We are fully aware of studies that extend Gaia in various facets such as expressiveness (Juan et al., 2002) and implementation (Moraitis et al., 2002). However, in this evaluation, we refer only to (Wooldridge et al., 2000b) and (Zambonelli et al., 2001b), written by the designers of the methodology.

3.1 Concepts and Properties

Below, we examine the extent to which Gaia addresses the concepts and the properties suggested by the evaluation framework. Gaia deals with all of the general concepts suggested, but lacks in depicting mental states of an agent.

- 1 **Autonomy:** in Gaia the autonomy is expressed by the fact that the role encapsulates its functionality (i.e., it is responsible for it). This functionality is internal and is not affected by the environment, thus represents the role's autonomy. In addition, in Gaia there is an option to model alternative computational paths, which gives the role (and agents that consist of this role) autonomy in making decisions. The ranking grade is 7.

- 2 Reactiveness: in Gaia the reactivity is expressed by the liveness properties within the role’s responsibilities. The ranking grade is 7.
- 3 Proactiveness: in Gaia the proactiveness is expressed by the liveness properties within the role’s responsibilities. The ranking grade is 7.
- 4 Sociality: in Gaia the sociality is expressed within the acquaintance model that defines the communication links among agent types. Further, some sociality aspects can be expressed using the organizational structure and rules. Yet, there is no explicit specification of relationships between organizations and roles and societies within MAS. The ranking grade is 4.

Examining the coverage of the framework’s building blocks by Gaia, we found that Gaia covers most of them, as seen in Table 7.1. However, the BDI concepts, the social building blocks, and the knowledge representation are not dealt with within Gaia. The ranking grade is 4.

Table 7.1. The coverage of the framework building blocks by Gaia

Framework building block	GAIA concepts
Agent	Agent type
Belief	
Desire	
Intention	
Message	Protocol
Norm	Organizational rule
Organization	Organization
Protocol	Protocol
Role	Role
Society	Organization
Task	Activity, Responsibility

3.2 Notations and Modeling Techniques

Following, we examine the extent to which Gaia addresses the notations and modeling techniques’ properties suggested by the evaluation framework. Gaia has room for improvements with regards to these properties. In addition, Gaia does not define its entire set of notations.



- 1 Accessibility: Gaia models are easy to understand and use. Yet, the behavior of the system is introduced via a set of logic expressions. This might introduce difficulties in understanding the behavioral specification of a system. The ranking grade is 5.
- 2 Analyzability: this issue is not dealt with within Gaia. The ranking grade is 1.
- 3 Complexity management: in Gaia, there is no hierarchical presentation or another mechanism for complexity management. The system's description is flat. The ranking grade is 1.
- 4 Executability: this issue is not dealt with within Gaia. The ranking grade is 1.
- 5 Expressiveness: Gaia is expressive and can handle a large variety of systems due to its generic structure. However, Gaia is mostly suitable for small and medium scale systems. This is because of its flatness, which limits the ability to model a large amount of details. In the following we present our analysis regarding the expressiveness of Gaia according to the properties defined in the previous section:
 - the structure of the system is not presented explicitly;
 - the knowledge encapsulated within the system is not presented explicitly;
 - the system's ontology is not dealt with;
 - the data flow within the system is depicted using textual specifications (via the dot and square brackets operators);
 - the control flow within the system is not presented explicitly;
 - the concurrent activities within the system (and within the agents) are not presented explicitly;
 - the resource constraints within the system (e.g., time, CPU and memory) are specified only partially using the permissions within Gaia;
 - the system's physical architecture is not dealt with;
 - the agents' mobility is not dealt with;
 - the interaction of the system with external systems are not presented explicitly; and
 - the user interface specification is not dealt with.

The ranking grade is 4.

- 6 Modularity: Gaia is mostly modular due to its design, including building blocks such as roles, protocols, activities and agent types. In Gaia, one can assign new roles to agents and remove roles with no effect on the internal model of the roles. However, changes within the protocol might cause changes within the internal structure of the role. These result in changes in permissions of the role, hence limits the modularity of Gaia. The ranking grade is 4.
- 7 Preciseness: the liveness and safety properties, which are used for depicting the functionality of a role in a formal way (i.e., for each symbol and notation there is a clear meaning and interpretation), make Gaia accurate and prevent misinterpretation of the modeled functionality. The symbols and notations of each of the other Gaia models have a clear meaning as well. The ranking grade is 7.

3.3 Development Process

Below, we examine the extent to which Gaia addresses the development process properties suggested by the framework. Gaia deals only with some aspects of the development process. With respect to the lifecycle coverage, it handles the analysis and design stages.

- 1 Development context: Gaia is adequate for the following development contexts: it can be used for creating new software, reengineering and designing systems with reuse components. However, Gaia does not support classical reverse engineering (from code to a model), since it does not address implementation aspects. For the same reason, it cannot be used for prototyping (especially, a rapid one). The ranking grade is 5.
- 2 Lifecycle coverage: is very limited within Gaia. It refers only to the analysis and the design stages within the development lifecycle. We found that fact a drawback of Gaia as a methodology, since it would require developers of MAS to adjust the Gaia-based design to the concepts of the target programming language. For example, one may translate the Gaia analysis and design results to UML notations and than use an object-oriented language for implementation. The ranking grade is 3.
- 3 Stages' activities within the methodology: Gaia provides a few guidelines for performing the analysis and design activities. The ranking grade is 4.
- 4 Verification and validation: these issues are not dealt with within Gaia. The ranking grade is 1.
- 5 Quality assurance: this issue is not dealt with within Gaia. The ranking grade is 1.

- 6 Project management guidelines: this issue is not dealt with within Gaia. The ranking grade is 1.

3.4 Pragmatics

Following, we examine the extent to which Gaia addresses the pragmatics properties suggested by the evaluation framework. Apparently, Gaia lacks in addressing some of these properties.

- 1 Resources: although Gaia is well known, there are not much available materials on it (except of the two cited papers). There are no users' groups, nor training or consulting services are offered. Additionally, Gaia does not provide automated tools. The ranking grade is 3.
- 2 Required expertise: Gaia requires a solid background and knowledge in logic and temporal logic. This reduces its accessibility since many developers do not know or do not want to get familiar with logic (and formal methods). The ranking grade is 4.
- 3 Language suitability: Gaia is not targeted at a specific language. It does not refer to the implementation issues, thus the specification made using Gaia can be implemented in any language. The ranking grade is 7.
- 4 Domain applicability: Gaia, as determined by its developers, is suitable to develop applications with the following characteristics: agents are coarse-grained computational systems, agents are heterogeneous, the abilities of the agents are static, and the number of agent types is comparatively small. Gaia is also suitable for dynamic-open systems where the agents are not known when designing the system. Yet, Gaia is not suitable for developing applications with dynamic characteristics such as goals generation. The ranking grade is 5.
- 5 Scalability: Gaia does not support the use of subsets thereof for system development. Yet, due to its simple structure, it may fit different application sizes. The ranking grade is 5.

4. Evaluating Tropos

In this section we evaluate Tropos (see chapter 5) according to the framework presented in section 7.2. In this evaluation we refer to Tropos as presented in (Castro et al., 2002) and (Giunchiglia et al., 2002).

4.1 Concepts and Properties

Below, we examine the extent to which Tropos addresses the concepts and the properties suggested by the framework. Tropos basically addresses the suggested properties, yet, it lacks in social-related aspects.

- 1 **Autonomy:** in Tropos the autonomy is specified in each one of the development phases. In the requirements phases of Tropos the autonomy is expressed by each actor's individual goals. Each actor has its own agenda of how to achieve its goals. In the architectural design phase the autonomy is expressed by the capabilities of each actor. These elaborate the actor's agenda for achieving the goals. In the detailed design phase the capabilities aforementioned are transformed into a set of UML activity diagrams, which specify the way in which the actor performs its activities. The ranking grade is 7.
- 2 **Reactiveness:** in Tropos the reactiveness is expressed explicitly only in some of its development phases. Reactiveness is appropriately addressed within the detailed design phase via the activity diagrams and the interaction diagrams. In the requirement phases of Tropos, events and responses to them are not expressed. In the architectural design phase, events and their responses are not specified as well. The ranking grade is 4.
- 3 **Proactiveness:** in Tropos the proactiveness is expressed by the activity diagrams. The ranking grade is 7.
- 4 **Sociality:** the sociality in Tropos focuses on the actor's interaction and dependencies. Yet, there are no means for modeling organizations and their actors. A study regarding the use of organization rules within Tropos was performed, however, it does not solve the hierarchical relationships and aggregation of actors and organizations. The ranking grade is 4.

Examining the coverage of the framework building blocks by Tropos, we found that it covers most of them, as seen in Table 7.2. However, the building blocks that are related to the social aspects are not addressed within Tropos. The ranking grade is 5.

4.2 Notations and Modeling Techniques

Following, we examine the extent to which Tropos addresses the notations and modeling techniques' properties suggested by the framework. Tropos is lacking in some of the important properties.

- 1 **Accessibility:** Tropos provides a set of models to support the entire development process. Each one of the models is simple to understand,

Table 7.2. The coverage of the framework building blocks within Tropos

Framework building block	Tropos concepts
Agent	Actor
Belief	Goal
Desire	Goal
Intention	Plan
Message	Activity diagram
Norm	Organization rule
Organization	
Protocol	Agent interaction diagram
Role	
Society	
Task	Capability

however, the need to synchronize among these models and the need to perform a model transformation throughout the development stages reduce its accessibility. Further, we could not find definition of transformation rules between the models. The ranking grade is 4.

- 2 Analyzability: Tropos handles the analyzability aspect via various development tools such as goal satisfaction (Fuxman et al., 2001) and model checking. Yet, it is unclear how consistency is checked among different stage of the development. The ranking grade is 4.
- 3 Complexity management: Tropos supports complexity management to a limited extent. In the requirement phases, one can look into the goal diagram of an actor, however, cannot control the level of details within. In the design phases the complexity management is supported by the UML mechanism. The ranking grade is 5.
- 4 Executability: Tropos demonstrates its executability using JACK (Howden et al., 2001). However, there is no discussion of what the efforts required to transform Tropos specification into JACK implementation are. Additionally, there is no discussion of other alternatives for implementation. The ranking grade is 4.
- 5 Expressiveness: Tropos mainly refers to BDI applications. In the following we present our analysis regarding the expressiveness of Tropos according to the properties defined in section 7.2:

- the structure of the system is not presented explicitly;
- the knowledge encapsulated within the system can be analyzed via the goal representations;
- the system's ontology is not dealt with;
- the data flow within the system is not presented explicitly;
- the control flow within the system is specified using the activity diagram within the detailed design phase;
- the concurrent activities within the system (and within the agents) are not presented explicitly;
- the resource constraints within the system (e.g., time, CPU and memory) are not dealt with; - the system's physical architecture is not dealt with;
- the agents' mobility is not dealt with;
- the interaction of the system with external systems is not presented explicitly; and
- the user interface specification is not dealt with.

The ranking grade is 4.

6 Modularity: within Tropos is fully supported. The ranking grade is 7.

7 Preciseness: the semantics within Tropos are clear and thus prevent mis-interpretation by its users. Tropos is formalized using the meta-model technique. The ranking grade is 7.

4.3 Development Process

Below, we examine the extent to which Tropos addresses the development process properties suggested by the framework. Tropos supports the traditional stages of software development, yet, it does not deal with the supportive activities.

- 1 Development context: Tropos is adequate for the following development contexts: it can be used for creating new software, reengineering, prototyping, and designing systems with reuse components. However, it does not support the classical reverse engineering (from code to a model), since when advancing from one stage to the preceding one several concepts undergo significant changes. The ranking grade is 6.
- 2 Lifecycle coverage: Tropos covers most of the lifecycle. However, it does not handle the testing stage. The ranking grade is 6.

- 3 Stages' activities within the methodology: the descriptions of the activities performed within the Tropos stages are somewhat lacking in the level of detail they provide. The ranking grade is 4.
- 4 Verification and validation: lately, Tropos was extended with a rich temporal specification language called Formal Tropos (Fuxman et al., 2001). Using that extension one may perform verification and validation over his/her application model. Yet, there is no coverage checking with respect to the initial requirements. The ranking grade is 5.
- 5 Quality assurance: this issue is not dealt with within Tropos. The ranking grade is 1.
- 6 Project management guidelines: this issue is not dealt with within Tropos. The ranking grade is 1.

4.4 Pragmatics

Following, we examine the extent to which Tropos addresses the pragmatics properties suggested by the framework. The Tropos community consists of many researchers and developers. This increases the amount of available resources.

- 1 Resources: Tropos has a lot of publications and many people supporting it. Additionally, a first workshop on Tropos has already been carried out. Yet, there are no users' groups, nor training or consulting services are offered. Tropos provides various automated tools for animation, model checking, and reasoning. The ranking grade is 5.
- 2 Required expertise: Tropos does not require special knowledge. The ranking grade is 7.
- 3 Language suitability: Tropos is based on the BDI concepts. Hence, its implementation will be biased towards that direction. The ranking grade is 4.
- 4 Domain applicability: Tropos is suitable for developing BDI-based applications, and is suitable for generic, componentized systems like e-business applications (Castro et al., 2002). The ranking grade is 7.
- 5 Scalability: Tropos does not provide details regarding the use of subsets or a superset thereof for system development. The ranking grade is 4.

5. Evaluating MaSE

In this section we evaluate Multiagent System Engineering (MaSE) (see chapter 6) according to the framework presented in section 7.2. In this evaluation we refer to work presented in (DeLoach, 2001; DeLoach, 2002; DeLoach et al., 2001; DiLeo et al., 2002; Self and DeLoach, 2003).

5.1 Concepts and Properties

Below, we examine the extent to which MaSE addresses the concepts and the properties suggested by the framework. MaSE basically addresses the suggested properties, yet, it lacks in social-related aspects.

- 1 **Autonomy:** in MaSE the autonomy is expressed by the fact that the role encapsulates its functionality. This functionality (i.e., its tasks) is internal and is not affected by the environment, thus represents the role's autonomy. The ranking grade is 7.
- 2 **Reactiveness:** in MaSE the reactiveness is not expressed explicitly. That is, there is no explicit connection between the event and the action taken. Yet, reactiveness can be expressed using the conversation state machines. The ranking grade is 4.
- 3 **Proactiveness:** in MaSE the proactiveness is expressed by the role's tasks. These tasks are modeled using finite state automata. The ranking grade is 7.
- 4 **Sociality:** in MaSE the social aspects of a system (except for communication) are not dealt with. MaSE does not provide means for gathering agents or defining organizations or societies. Yet, some sociality aspects can be expressed using organizational rules. The ranking grade is 4.

Examining the coverage of the framework building blocks by MaSE, we found that MaSE addresses most of them, as seen in Table 7.3. MaSE conceptualizes a MAS, as a set of agents, however, there is no higher level of abstraction such as an organization or a society. Nevertheless, it provides means for defining organization rules. The BDI concepts can be expressed using the goal, the task and the state building blocks within MaSE. The message and protocol building blocks are expressed in MaSE within the conversation diagrams and the task diagrams; however, MaSE does not explicitly show the communication pattern. The ranking grade is 5.

5.2 Notations and Modeling Techniques

Following, we examine the extent to which MaSE addresses the notations and modeling techniques' properties suggested by the framework. MaSE ad-

Table 7.3. The coverage of the framework building blocks within MaSE

Framework building block	MaSE concepts
Agent	Agent type
Belief	Goal, Task, State
Desire	Goal, Task, State
Intention	Goal, Task, State
Message	Conversation
Norm	Organization rule
Organization	
Protocol	Conversation, Task
Role	Role
Society	
Task	Task

addresses most of these properties to a satisfactory level. However, it could still benefit from further improvements.

- 1 Accessibility: MaSE provides a very simple set of models which enhance accessibility, however, the need of synchronizing among these models and the need to perform a model transformation throughout the development stages reduce its accessibility. The ranking grade is 5.
- 2 Analyzability: MaSE supports consistency checking and internal verification of the models. However, there are still some cases where inconsistencies may occur, for example between a sequence diagram and the transition between the states in a conversation. The ranking grade is 6.
- 3 Complexity management: There are several layers of abstraction within MaSE: agents, roles, and tasks. However, there is no support of managing the complexity of complex tasks and roles, e.g., there are no means for describing composite tasks. In addition, MaSE does not allow defining a role hierarchy. The ranking grade is 4.
- 4 Executability: MaSE supports partial code generation using agentTool (see chapter 6), which is the CASE tool that supports the MaSE methodology. The generation includes complete conversations for many communication frameworks. Since, MaSE dynamics is based on finite state

automaton it has the potential to achieve a code generation of high quality. The ranking grade is 4.

5 Expressiveness: MaSE has been used in several systems such as the Multi-Agent Distributed Goal Satisfaction (MADGS) and the course scheduling system. In the following we present our analysis regarding the expressiveness of MaSE according to the properties defined in section 7.2:

- the structure of a system is explicitly described using the agent architecture and system design diagrams;
- the knowledge encapsulated within the system is not presented explicitly;
- the system's ontology is fully supported;
- the data flow within the system cannot be specified explicitly;
- the control flow within the system is not presented explicitly, however, it can be understood from the concurrent task diagram set;
- the system's physical architecture is specified using the deployment diagram;
- the agents' mobility is dealt with within MaSE via a special method (i.e., the method move);
- the interaction of the system with external systems is specified using the conversation concept; and
- the user interface specification is not dealt with, however, MaSE recommends that the user interface will be treated as a separate role.

The ranking grade is 5.

6 Modularity: within MaSE is supported within the agent template diagram. However, reuse of elements within MaSE such as tasks, roles, protocols and conversations is not supported. The ranking grade is 4.

7 Preciseness: the semantics within MaSE are clear and thus prevent misinterpretation by its users. However, MaSE does not provide formal definitions of its notations and models. The ranking grade is 6.

5.3 Development Process

Below, we examine the extent to which MaSE addresses the development process properties suggested by the framework. In general, MaSE addresses the traditional stages within the development process, however, it lacks in addressing the supportive activities.

- 1 Development context: MaSE is adequate for the following development contexts: it can be used in creating new software, reengineering and designing systems with reuse components, and prototyping. However, MaSE does not support reverse engineering, since it may be difficult to transform models backwards. The ranking grade is 5.
- 2 Lifecycle coverage: is comprehensive within MaSE. The goal capturing sub-stage and the applying use cases sub-stage can be considered as a requirement stage, the analysis stage consists of the roles and tasks definition sub-stage and building the system ontology sub-stage, the design stage consists of the construction and assembling of agents sub-stage and the implementation stage consists of the system design sub-stage and the code generation. The testing stage is not covered by MaSE. The ranking grade is 5.
- 3 Stages' activities within the methodology: MaSE provides guidelines for performing the activities of the development stages. The ranking grade is 7.
- 4 Verification and validation: MaSE performs verification over its models to check consistency, to identify deadlocks and unused elements. In addition, it provides guidelines that support the coverage checking between the stages. Yet, MaSE does not provide guidelines or means for checking the requirements against the outcomes of each one of its stages. The ranking grade is 4.
- 5 Quality assurance: this issue is not dealt with within MaSE. The ranking grade is 1.
- 6 Project management guidelines: this issue is not dealt with within MaSE. The ranking grade is 1.

5.4 Pragmatics

Following, we examine the extent to which MaSE addresses the pragmatics properties suggested by the framework. MaSE provides a solid infrastructure that encourages its use.

- 1 Resources: MaSE has many publications. It also has a Web site and a CASE tool (agentTool). Yet, there are no users' groups, nor training or consulting services are offered. The ranking grade is 5.
- 2 Required expertise: MaSE requires a solid background and knowledge in logic and temporal logic for using the organization rules. Other models do not require specific knowledge except for finite state automata. The ranking grade is 5.

- 3 Language suitability: MaSE is not targeted at a specific programming language, a specific architecture, or a specific framework. The ranking grade is 7.
- 4 Domain applicability: MaSE is a general-purpose methodology for designing MAS. The designers of MaSE report on using MaSE for various types of agents and domains. The ranking grade is 7.
- 5 Scalability: MaSE does not provide details regarding the use of subsets or a superset thereof for system development. It seems that when using MaSE for specifying a system of any size, its user must follow the exact development path set by MaSE. The ranking grade is 3.

6. Summary and Conclusion

We have evaluated three agent-oriented methodologies, utilizing a framework that examines the various facets of a methodology. The results of that evaluation are summarized in Table 7.4. Examining the table, it appears that the level to which the evaluated methodologies support the various facets is, in general, medium to high, and this is a positive result. Yet, there is space for further improvements. Below we summarize the evaluation.

- 1 Concepts and properties: the autonomy and proactiveness criteria are properly addressed by the methodologies. The reactivity is lacking in the sense that the connection between events and responses to them is not well specified. The sociality deals with organization rules, but not with multiple organizations or societies structure, nor with role hierarchy. The building blocks coverage is good, however none of the methodologies covers them all. This coverage varies among the methodologies as a result of their different goals.
- 2 Notations and modeling techniques: this facet is addressed to a limited extent. The accessibility of the methodologies is fine, however requires further enhancements. Current limitations result from the multiplicity of models and the use of logic within the specification stages. The analyzability property is addressed within MaSE and Tropos. Complexity management is hardly addressed within Gaia. Tropos and MaSE need further enhancements in this respect as well. The level to which the methodologies support executability is unclear. The expressiveness is lacking in terms of multiple modeling needs not being explicitly addressed, e.g., the system knowledge and ontology, the data and control flow and the user interface. The support for modularity requires further improvements. The preciseness property is well supported by the methodologies.

- 3 Development process: this facet is lacking in both quality assurance and project management. The methodologies support various development contexts. With respect to the stages' activities, MaSE provides a detailed description of the development activities. The other two methodologies require further enhancements. In addition, the traceability between the different stages is unclear in all of the methodology.

Table 7.4. Methodologies evaluation summary

Framework Criteria	GAIA	Tropos	MaSE
Concepts and properties			
Autonomy	7	7	7
Reactiveness	7	4	4
Proactiveness	7	7	7
Sociality	4	4	4
Building blocks coverage	4	5	5
Notations and modeling techniques			
Accessibility	5	4	5
Analyzability	1	5	6
Complexity Management	1	5	4
Executability	1	4	4
Expressiveness	4	4	5
Modularity	4	7	4
Preciseness	7	7	6
Development process			
Development context	5	6	5
Lifecycle coverage	3	6	5
Stages activities	4	4	7
Validation and verification	1	5	4
Quality assurance	1	1	1
Project management	1	1	1
Pragmatics			
Resources	3	5	5
Required expertise	4	7	5
Language suitability	7	4	7
Domain applicability	5	7	7
Scalability	5	4	3

- 4 Pragmatics: Gaia and MaSE suffer from lack of resources and very limited use. Tropos provide many resources since it is developed in several research institutes. Except for MaSE, the methodologies lack in providing environments to support the development stages. Except for logic,

the methodologies do not require that designers have special expertise for their use. The methodologies are not coupled to a specific programming language or an agent architecture, and can be used for multiple domains. Scalability (i.e., the ability to be adjusted according to a specific project needs) is not supported by the methodologies.

In conclusion, the examined agent-oriented methodologies provide an appropriate infrastructure, however there is a need for further research and improvements. This is an important conclusion in support of agent-oriented methodologies, as it may promote these enhancements and help arriving at industry-grade methodologies. Additionally, the evaluation performed here provides researchers and practitioners with a detailed comparison among the leading agent-oriented methodologies. Further, the framework used in this study may be utilized by others to evaluate and compare other methodologies as needed.

III

SPECIAL-PURPOSE METHODOLOGIES

Introduction

A general methodology is a methodology applicable to a wide range of MAS, i.e., not devoted to a specific agent architecture. A special-purpose methodology is a methodology which is dedicated to a field of applications such as Internet applications, telecommunication applications, e-commerce applications, etc. and sometimes as a result to a specific computational architecture agent and/or platform.

Applications belonging to a same field share the same characteristics which are taken into account in the methodology and consequently are predominant to choose the methodology. It is quite evident that the characteristics of the applications are also those that are required by an agent or MAS. But in addition, the concerned applications have well-know and well established characteristics such as: the system's environment is dynamic (making it ineffective to exhaustively enumerate all the situations the system may encounter), the system is open and therefore dynamic because it is constituted of a shifting number of components, the task the system has to achieve is so complex that a perfect design cannot be guaranteed, the system must be implemented on the Internet, agents in the system need negotiation, etc.

A general methodology is suitable to develop the associated software but using a special-purpose methodology improves the software development. It facilitates designers' work and shortens the development time. In these methodologies, the tasks to be realized in the requirement and analysis phases are quite identical to the tasks to do in all methodologies. The differences are more significant in the three last phases, i.e., design, implementation and deployment. The design phase is generally more detailed than the general ones because the agent architecture is known. That is to say the different modules which are composing an agent must be filled up during a step of the design phase. For example, if the agent architecture is BDI, designers have to explicitly describe beliefs, desires and intentions. The design of each of these modules is specified during the design phase. So the design phase of these methodologies is

detailed and leads easily to implementation. Because sometimes the platform to develop the software is known, development and deployment phases are made easier for designers. For example, having an automatic code generation towards a chosen platform would be conceivable.

The four chapters in this part explore four methodologies: ADELFE, MESSAGE, SADDE and Prometheus. In a sense, having classified these methodologies as special-purpose ones is not intended to support the claim that they are not general enough, but rather to be explicit on the fact that they are possibly much more suited than others for specific aspects.

- Chapter 8, “The ADELFE Methodology” written by Gauthier Picard and Marie-Pierre Gleizes expounds this methodology, which is dedicated to applications characterized by openness and the need of the system adaptation to an environment. This methodology is based on the RUP process and follows its main phases: preliminary requirements, final requirements, analysis and design, and uses UML and AUML notations. Some of the new tasks compared with object-oriented methodologies are: the characterization of the environment in the final requirements, a step to decide if an adaptive system is needed to design the application and agent identification during the analysis phase. During the design phase, a specific architecture of cooperative agent is used to design agents. A commercialized tool “OpenTool” supporting UML was extended to support AUML notations.
- Chapter 9, “The MESSAGE Methodology” by Giovanni Caire, Win Coulier, Francisco Garijo, Jorge Gomez, Juan Pavon, Paul Kearney and Philippe Massonet, describes this methodology, developed for mainstream software engineering departments and for designing telecommunications applications. The main concepts of MESSAGE are agent, organization, role, task and interaction. MESSAGE follows the RUP process and uses UML and AUML notations. Authors are focusing the presentation on the analysis and the design phases. The methodology offers designers some refinement strategies in which each level expands concepts of the lower levels. The design phase is divided into two parts: a higher level which refines analysis models and a lower level which deals with a BDI agent architecture or JADE or FIPA-OS platform.
- Chapter 10, “The SADDE Methodology” by Carles Sierra, Jordi Sabater, Jaurne Agusti and Pere Garcia presents the general steps of SADDE methodology and its use in electricity market case study. This methodology is dedicated to Electronic Institutions applications. Four main steps are composing SADDE: the Equation-Based Model to model the global behavior of the system, the Electronic Institution Model to specify interactions between agents, the Agent-Based Model which uses evolutionary

computing and the MAS level to experiment. After the last step, designers can realize backtracks to redesign and refine the different previous models of the system.

- Chapter 11, “The Prometheus Methodology” by Michael Winikoff and Lin Padgham gives an overview of Prometheus methodology which is based on the RUP and is composed of the specification, the architectural design and the detailed design phases. During the specification phase, interactions between the system and the environment are studied in terms of percepts and actions. During the architectural design phase, designers can identify agent types by grouping some functionality. During the detailed design phase, agents are designed using an iterative refinement from their functionalities, at the beginning BDI agent architectures are used. Prometheus Design Tool, the tool associated with the methodology, supports the process of the methodology, enables to design diagrams and to check some inconsistencies. JACK Development Environment supports Prometheus and enables generating a skeleton code from design artifacts.

Chapter 8

THE ADELFE METHODOLOGY

Gauthier Picard and Marie-Pierre Gleizes

Abstract This paper presents a method named ADELFE, which is led by the Rational Unified Process but is devoted to software engineering of adaptive MAS. ADELFE guarantees that the software is developed according to the AMAS theory¹. We focus this presentation on the four first core workflows of the RUP. Therefore, in the preliminary requirements an agreement on what the system has to do must be reached. During the final requirements phase, the environment of the studied system must be defined and characterized. Then, in the analysis phase, the engineer is guided to decide to use adaptive multiagent technology and to identify the agents through the system and the environment models. Finally, the design workflow of ADELFE must provide the cooperative agent's model and helps the developer to define the local agents' behavior. We illustrate the methodology by applying it to a case study: a timetable design.

1. Introduction

Nowadays, problems to solve in computer science are becoming more and more complex (like information search on the Internet, mobile robots moving in the real world and so on). Systems able to respond to such problems are open and complex because they are incompletely specified, they are immersed in a dynamical environment and more importantly an a priori known algorithm to find a solution does not exist. This solution must build itself according to interactions the system will have with its environment during its functioning. Classical approaches to solve problems in such a context cannot be applied. That led us to propose a theory called AMAS (Adaptive Multi-Agent System) theory (Capera et al., 2003a) (see chapter 16), based on the use of self-organizing systems (see chapters 17).

This theory has been successfully applied to many projects: a tool to manage the knowledge required to assist a user during information retrieval training, an

¹The Adaptive Multi-Agent Systems (AMAS) theory has been in development and applied for the last 8 years at the Research Institute in Computer Science of Toulouse (IRIT).

electronic commerce tool for mediation of services, a software tool for adaptive flood forecast or adaptive routing of the traffic in a telephone network. Obtained results led us to promote the use of self-organizing systems based on the AMAS theory and to build a method for designing such systems. They are required both to reuse our know-how and to guide an engineer during an application design. In that sense, ADELFE is a toolkit to develop software with emergent functionality (Bernon et al., 2002). ADELFE is not a general method; it concerns applications in which self-organization makes the solution emerge from the interactions of its parts. It guarantees that the software is developed according to the AMAS theory. It also gives some hints to the designer to tell him if using the AMAS theory is relevant to build his application. The ADELFE toolkit enables the development of software with emergent functionality and consists of a software development process, a notation based on UML/ AUML (see chapter 12), some tools supporting the process, and the notations and a library of components that can be used to make the application development easier.

This chapter is structured as follow: section 2 gives principal concepts of the AMAS theory and a brief overview of the methodology and of the case study: ETTO (Emergent Time Tabling Organization) application used to illustrate the process. Then, sections 3, 4, 5 and 6 detail respectively the requirement, the specification and the design work definitions. Section 7 presents the main tools associated with ADELFE. Before concluding, section 8 gives a brief comparison to some other well-known methodologies.

2. ADELFE Methodology Overview

In this section, after a brief presentation of the AMAS theory on which ADELFE is based on, an overview of the ADELFE method is expounded. Then, the requirements for the timetabling problem² case study used to illustrate the methodology are presented.

2.1 The AMAS Theory

The AMAS theory provides a solution to build complex systems for which classical algorithmic solutions cannot be applied (Capera et al., 2003a; Capera et al., 2003b). Concerned systems are open and complex. All the interactions the system may have with its environment cannot be exhaustively enumerated; unpredictable interactions can occur during the system functioning and the system must adapt itself to these unpredictable events. The solution provided by the AMAS theory is then to rid ourselves of the global searched goal by build-

²This example was elaborated as a case study to compare and discuss different methodologies and multi-agent platforms for the ASA Group of the French Artificial Intelligence Association.

ing artificial systems for which the observed collective activity is not described in any agent composing it. Each internal part of the system (agent) only pursues an individual objective and interacts with agents it knows by respecting cooperative techniques which lead to avoid unpredictable situations (like conflict, concurrency, etc.), called Non Cooperative Situations (NCS). Faced with a NCS, a cooperative agent acts to reach a new cooperative state and permanently adapts itself to unpredictable situations while learning on others. Interactions between agents depend on their local view and on their ability to cooperate with each other. Changing these local interactions reorganizes the system and thus changes its global behavior.

Applying the AMAS theory consists in enumerating, according to the current problem to solve, all the cooperative failures that can appear during the system functioning and then defining the actions the system must apply to come back to a cooperative state.

2.2 ADELFE Overview

The ADELFE process consists in six work definitions: Preliminary Requirements, Final Requirements, Analysis, Design, Implementation and Tests.

Among the different activities or steps that are listed in Figure 8.1, some are marked with a bold font to show that they are specific to adaptive MAS. Only the four work definitions require modifications in order to be tailored to AMAS and the main activities are presented in the next sections. Other work definitions appearing in the RUP remain the same (Jacobson et al., 1999).

2.3 The ETTO Case Study

In order to show how a self-organizing application can be developed using the tools linked with ADELFE, the next sections will refer to the ETTO (or Emergent Time Tabling Organization) application. Description and design of the system related to ETTO are not the main objective of this article and more information is available in (Bernon et al., 2002). The chosen problem is a classical course timetabling one in which time slots and locations (rooms) must be assigned to teachers and students groups in order to let them meet during lectures. Usually, solutions to such a problem can be found using different techniques like constraint-based ones or meta-heuristics techniques (simulated annealing, taboo search, graph coloring, etc.) and more recently, neural networks, evolutionary or ant algorithms. However, no real solving technique exists when the constraints can dynamically evolve and when the system needs to adapt. Because this problem is not a simulation one, because it is actually complex when handmade or not (it belongs to the NP-complete class of problems) and has no universal solution, we do think that it represents the right example to apply the AMAS theory. The aim is to make a solution emerge, at

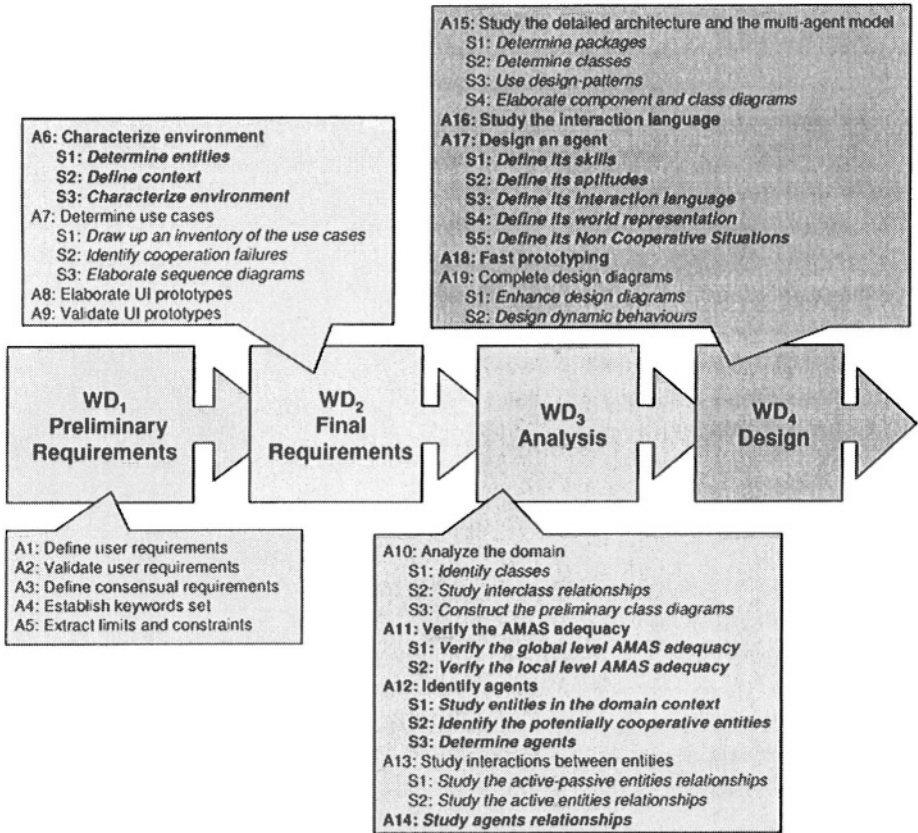


Figure 8.1. ADELFE process is described in three levels: Work Definitions (WD_{*i*}), Activities (A_{*j*}) and Steps (S_{*k*})

the macro-level of the built MAS, from the interactions of independent parts at the micro-level.

General requirements for this ETTO problem are the following. Stakeholders are teachers, students groups and lecture rooms. Every actor individually owns some constraints that must be (at best) fulfilled. A teacher has some constraints about his availabilities (e.g., the days or the time slots during which he can teach), his capabilities (e.g., the topics he can lecture on) and the needs he has about particular pedagogic equipments (overhead projectors, video projectors, a defined lecture room for a practical work, etc.). A students group must take a particular teaching made up of a certain number of time slots for a certain number of teaching topics (X time slots for a topic 1, Y time slots for a topic 2, etc.). A lecture room is equipped or not with specific equipments (an overhead projector, a video projector or any equipment for practical works) and can be occupied or not (e.g., during a given time slot or on a certain day).

3. Preliminary Requirements

The aim of the preliminary requirements is to define the system to be and to establish an agreement on the preliminary requirements.

The preliminary requirements work definition concerns the description of the system and the environment in which the system will be deployed. It consists in defining what to build or what is the most appropriate system for end-users. End-users, clients, analysts and designers have to list the potential requirements, to define the context in which the system will be deployed and to list the functional and non-functional requirements (Activity #1 and #3). They must agree on these requirements (Activity #2 and #3). Then, designers have to define the main concepts used to describe the application and its domain (the system and its environment) (Activity #4). And they must define the limits and constraints of the system they have to build (Activity #5).

4. Final Requirements

The aim of the final requirements is to transform this view in a use-case model, and to organize and to manage the requirements (functional or not) and their priorities. In fact, at this stage, the designer has to define the function of the studied system and to model its environment. To take into account adaptive MAS, four steps are added to the RUP process: three are in the Activity #6 (Characterize environment) and one is in the Activity #7 (Determine Use cases). The two last activities in this workflow relating to User Interface elaboration are not described here.

4.1 Characterization of the Environment

This activity (Activity #6 in Figure 8.1) is divided into three steps: determining the entities that are involved in the system, defining the context and characterizing the environment. Entities to identify are active or passive entities in interaction with the system.

A detailed definition of the system environment is necessary to develop adaptive systems, which are able to respond to any change. This step firstly focuses on what may be in interaction with the studied system in terms of passive or active entities, or constraints. In our example, teachers, students, the planning manager and the room manager are active entities because they are able to change by themselves their own constraints or they can interact with the system. Rooms are passive because they represent resources and they cannot modify their characteristics by themselves. The NPP (or National Pedagogic Plan) is the database that contains all information concerning the courses (maximum number of sessions per week, hour quotas for each formation, etc): it is a passive entity.

In a second time, this step must define the context of the system. It requires a characterization of data streams and interactions between entities and the system. Data streams between passive entities and the system are expressed using collaboration diagrams. Interactions between active entities and the system are expressed using sequence diagrams.

In our example, two kinds of data flows between the system and passive entities exist: when the system consults the NPP and when the system consults room constraints. When an active entity wants to interact with the system, it may only have to change constraints (owner constraints or room constraints). In the other sense, the system interacts with the active entity by displaying the planning.

Finally, to characterize the environment, designers must think about the environment of the system to build in terms of being accessible or not, deterministic or not, dynamic or static and discrete or continuous. These terms have been reused from (Russell and Norvig, 1995) and represent a help to later determine if the AMAS technology is needed or not. This characterization may enable the designer to detect some special use cases to respond to environment behavior. In the case study, the environment of the system can be characterized as follow:

- *Dynamic*: the evolution of the active entities does not depend on the system, they are unpredictable from the point of view of the system;
- *Accessible*: the environment can obtain information on the state of the environment;
- *Non-deterministic*: the system is not able to know what could be the effects of its actions on active entities; and
- *Continuous*: the number of interactions between system and entities are infinite.

4.2 Determination of the Use Cases

The main objective of this activity, which ends the requirements workflow, is to clarify the different functionalities the system has to respond to. This activity is divided into three steps which enable to design use cases, to elaborate the associated sequence diagrams and to identify cooperation failures. Only active entities are implied in these use cases, which are the results of a functional requirements set. Identification of cooperation failures between the system and its environment is realized in order to help designers to detect problems in the sense of the AMAS theory: Non Cooperative Situations. This identification will be refined during the development process and enables identification of agents later in the process. The use cases for the timetabling problem are shown

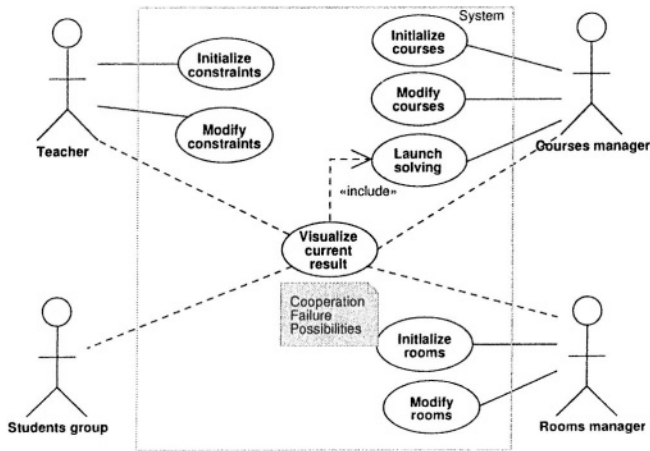


Figure 8.2. The use cases for the ETTO problem. Associations to Visualize current result case are potentially non cooperative: the result of the time tabling resolution is the only cause of cooperation failure between the users and the system, in the sense that users expect the system to satisfy their constraints

in Figure 8.2. Cooperation failures are represented on use cases diagrams by dotted lines.

5. Analysis

From a multiagent point of view, the identification of agents must take place in this workflow. The analysis work definition has to develop an understanding of the system, its structure in terms of components and to know if the AMAS theory is required.

5.1 Domain Analysis

Domain analysis (Activity #10) is a static view and an abstraction of the real world and the linked entities. Considering separately each use-case by defining scenarios, the designer has to divide the system into entities. The result of this step is a set of entities in preliminary class diagrams. Teacher, CourseManager, StudentGroup, Room, RoomManager and NPP classes appear naturally as real world entities. In a second time, we tried to determine what entities could be useful for our system. A board is proposed to visualize the organization (Grid and Cell classes) and the ConstraintManager class to control constraints for each entity that owns a Constraint class instance. Cells represent intersections of different dimensions (days, rooms, etc).

5.2 Adequacy of the AMAS Theory

This activity (Activity #10 in Figure 8.1) aims to help the designer to decide if the AMAS theory is adequate to solve his problem because, for certain applications, this kind of programming can be useless. A software component has been developed with several criteria to study the adequacy at two levels:

- At the global level to answer the question “is a system implementation using AMAS needed?”
- At the local level to try to answer the question “do some components need to be implemented as AMAS?” i.e., is some decomposition or recursion useful during design?

For the case study, the decision tool clearly suggests to use the AMAS to design the global level. Moreover, the tool indicates that some entities could be decomposed as AMAS. So, once the agents are identified, the designer has to reuse the method on them, as developed below.

5.3 Agent Identification

In this activity (Activity #12), we are only interested in agents that enable a designer to build our sort of AMAS. The designer has to determine which entities fit with this agent type: “cooperative agents.” A cooperative agent ignores the global function of the system; it only pursues an individual objective and tries to be permanently cooperative with other agents involved in the system. The global function of the system is emerging (from the agent level to the multiagent level) thanks to these cooperative interactions between agents. An agent can also detect Non Cooperative Situations (NCS) that are situations it judges as being harmful for the interactions it possesses with others, such as

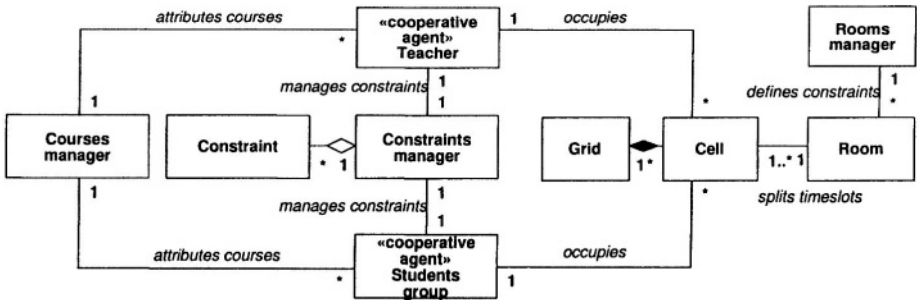


Figure 8.3. The main class diagram for the ETTO problem. Three classes of agents appear: the StudentsGroup, the Teacher and the BookingAgent. The two firsts are interface agents between the system and the users. BookingAgents aim to reserve time slots (Cell of a Grid) and Rooms for Teachers or StudentsGroups in terms of their Constraints

lack of understanding, ambiguity or uselessness. This does not mean that an agent is altruistic or always helping other agents but it is just trying to have useful (from its point of view) interactions with others. Thus, facing up to a NCS, an agent always acts to come back to a cooperative state. In fact, the behavior of the collective compels the behavior of an agent. In ADELFE, designers are given some guidelines: an entity can become an agent if it can be faced with “cooperation failures” and may have evolutionary representations about itself, other entities or about its environment and/or may have evolutionary skills. At this stage, we identify teachers and students groups as being cooperative agents. All other entities are considered as objects.

5.4 Adequacy of the AMAS Theory at the Local Level

If the first step of adequacy to the AMAS theory indicates a possible decomposition, each agent has to be analyzed as a system. The goals of an agent, Teacher or StudentGroup, are to find different places and partners to follow or to give each course. These goals raise the problem of ubiquity. Agents cannot be at different places at different moments. Therefore, we propose to create one agent per course for each teacher or student group. Two agent levels are distinguished:

- RepresentativeAgent (RA): at the highest level, it represents a teacher or a student group within the system;
- BookingAgent (BA): at the lowest level, it is responsible for finding partners and booking rooms for a RA. There are as many BA as the number of courses a teacher has to give or a student group has to follow.

The identified agents have to be added to the preliminary class diagram as shown in Figure 8.3.

6. Design

The design work definition aims to formulate models that focus on non functional requirements and the solution domain and that prepare for the implementation and test of the system. In ADELFE, agents being identified and their relationships being studied, designers have now to study the way in which the agents are going to interact (Activity #15) thanks to protocol diagrams. ADELFE also provides a model for designing cooperative agents (Activity #16); designers must describe, for each type of agents, its skills, its aptitudes, its interaction language, its world representation and the Non Cooperative Situations this agent can encounter. The global function of a self-organizing system is not coded; designers have only to code the local behavior of the parts composing it. An activity of fast prototyping (Activity #17) based on finite state machines has been added to the process. It enables designers to verify the be-

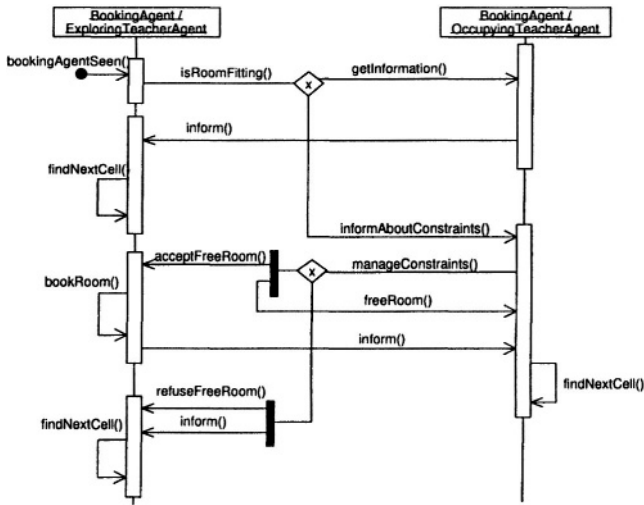


Figure 8.4. An example of protocol diagram between two BookingAgents of two different Teachers. The first agent explores the timetable grid to find satisfying slots and rooms. The second one already occupies a room and a slot. This diagram explains the negotiation between these two agents when the first agent meets the second agent. This negotiation may either result on the leaving of the first agent or the booking, by the first agent, and the leaving of the second agent

havior of the agents being built. Then designers have to complete the previous defined class diagram (Activity #18). Once a class diagram enhanced, finalized indeed, this diagram may require the development of a statechart diagram. The aim is to highlight the different state changes of an entity when it is interacting with others.

Because the complete design cannot be described in this chapter, we only detail the agent design activity which does not exist in other methodologies. Five steps compose this activity, the fourth one enables to endow an agent with classical parts such as: skills, aptitudes, an interaction language and world representations and the last one is more specific to AMAS theory.

6.1 Study of Interaction Languages

The result of this activity (Activity #15) is a set of protocol diagrams representing the different interaction languages that may be used by the agents. Figure 8.4 shows a sample protocol between two BookingAgents having two different roles: ExploringTeacherAgent and OccupyingTeacherAgent. The AUML – Agent Unified Modeling Language – (see chapter 12) notation is used but some specific functionality has been added to AIP diagrams to fit to the AMAS theory requirements. The decision-making process corresponding to an OR or

a XOR branch is done by an «aptitude»-stereotyped operation attached to the branch-node (see later in this section). For example, in Figure 8.4, the isRoom-Fitting operation is attached to the first XOR node; i.e., depending on the results of this operation, the ExploringTeacherAgent may either request for more information before resuming its exploration of the grid, or negotiate in terms of the constraints which agents own.

Once the set of protocols defined, designers may assign them to agents during the new activity, as these are generic. Another possibility is to specify fully generic protocols in which only roles are manipulated.

6.2 Agent Design

This activity helps the designer to fill in a generic architecture given for an agent used in the AMAS theory. ADELFE is a method which is devoted to a specific kind of agents: cooperative ones. Therefore, even if an agent still follows the same defined life cycle – it gets perceptions from its environment and autonomously uses them to decide what to do in order to reach its own goal and, finally, acts to realize the action it has decided before – it has some specific characteristics and is then composed of five parts that will constitute its own behavior:

- *Skills* that are knowledge about a domain enabling the agent to perform actions.
- *Aptitudes* which are the abilities an agent possesses to reason on its knowledge (concerning the domain) or on its representation of the world.
- An *interaction language* which enables the agent to interact and communicate with others in a direct or indirect (possibly, using its environment) way.
- *Representations* of the world that are knowledge used by an agent to represent itself, other agents or its environment.
- *Non Cooperative Situations* that an agent must detect and process because these situations are judged “harmful” for both the agent and its viewpoint about the collective.

ADELFE Stereotypes. To enable the developer to deal with these specific components in the ADELFE methodology, nine stereotypes have been defined to express how an agent is formed and/or how its behavior may be expressed: «cooperative agent», «Characteristic», «perception», «action», «skill», «aptitude», «representation», «interaction» and «cooperation».

In order to modify the semantics of classes and features depending on the specificities of cooperative agents these stereotypes and their rules (written in

OTScript language) are included in the OpenTool graphical tool linked with ADELFE. All these stereotypes, except «cooperative agent», can be applied to attributes and/or methods. All the examples appearing in this section refer to Figure 8.5.

The «cooperative agent» stereotype expresses that an entity is an agent which has a cooperative attitude and can be used to build AMAS. An agent will be implemented using a class that will be stereotyped with «cooperative agent». This class must have a run method that simulates the agent’s life cycle. Therefore, to ensure that this method does exist, an agent inherits from a superclass called CooperativeAgent. A sample associated coherency rule is: an agent-stereotyped class inherits (directly or not) from the CooperativeAgent class. For example, in the course timetabling application, BookingAgents (BA) have been identified to represent teacher and/or student entities. A BA’s goal is to find convenient time slots in the timetable. A BookingAgent class can then be defined and stereotyped with «cooperative agent». This class inherits from the CooperativeAgent class and therefore contains four methods: run, perceive, decide and act.

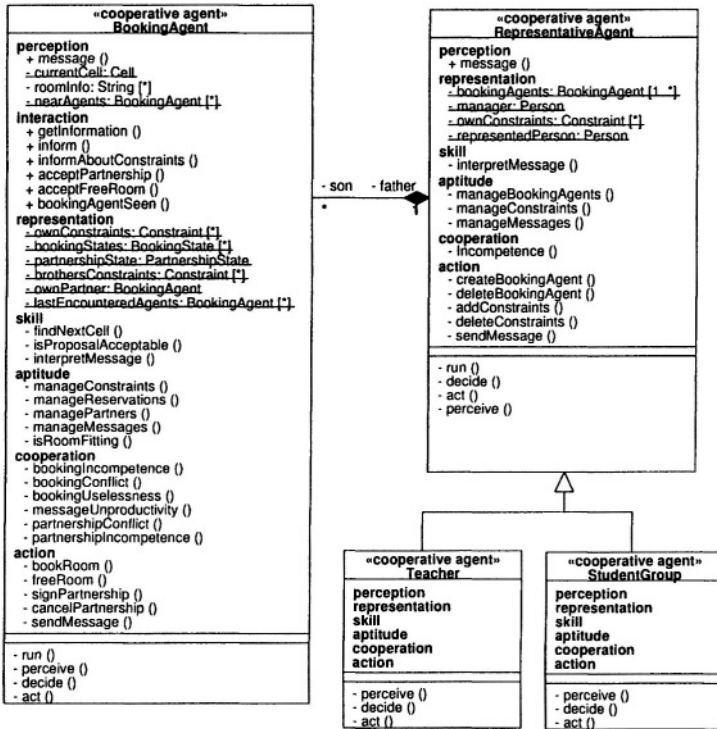


Figure 8.5. The two main «cooperative agent»-stereotyped classes: RepresentativeAgent and BookingAgent. The first one can represent either a StudentGroup or a Teacher

The «Characteristic» stereotype is used to tag an intrinsic or physical property of a cooperative agent. An attribute represents the value of a property. A method modifies or updates the value of a property. A characteristic can be accessed or called anytime during the life cycle. It can also be accessed or called by other agents.

The «perception» stereotype expresses how an agent receive information from the physical or social (other agents) environment. Attributes represent data coming from the environment. Methods are means to update or modify «perception»-stereotyped attributes. A associated coherency rule is: “an attribute stereotyped with «perception» is necessarily private.”

The «action» stereotype is used to signal how an agent acts on the environment during its action phase. Methods are possible actions for an agent. Attributes are parameters of an action. An agent is the only one that can use its actions. A coherency rule associated is: “an attribute stereotyped with «action» is private and a method that is stereotyped using «action» is private and can only be called during the action phase of an agent.”

The «skill» stereotype is used to tag specific knowledge enabling an agent to realize its own partial function. Methods represent reasoning an agent can do. Attributes are data useful to act on the world or parameters of a «skill»-stereotyped method. Such an attribute or method can only be accessed/affected or called by the agent itself to express its autonomy of decision. Skills can be represented by a MAS when they need to evolve. A coherency rule associated is: “an attribute or a method that is stereotyped with «skill» is necessarily private. Such an attribute can only be used by a «skill»-stereotyped method.”

The «aptitude» stereotype expresses the ability of an agent to reason both about knowledge and beliefs it owns. Methods express reasoning that an agent is able to do. Attributes represent functioning data or parameters of reasoning. A method or an attribute which is stereotyped with «aptitude» can only be accessed, or affected, or called by the agent itself, to express its autonomy. Coherency rules associated are:

- An attribute or a method that is stereotyped with «aptitude» is necessarily private.
- An «aptitude» attribute can only be used by a method that is also stereotyped with «aptitude».
- A method that is stereotyped with «aptitude» can only be called during the decision phase of the agent.
- An «aptitude»-stereotyped method can only call methods or attributes that are stereotyped with «perception», «representation» or «interaction».

The «representation» stereotype is a means to indicate world representations that are used by an agent to determine its behavior. Attributes are knowledge

units describing an agent. Methods are means to handle representations: access or alteration. Representations that may evolve can be expressed using a MAS. Coherency rules associated are:

- An attribute or a method which is stereotyped with «representation» is necessarily private.
- A «representation»-stereotyped attribute can only be used by a method which is stereotyped with «representation» or «aptitude».
- A «representation»-stereotyped method can only be called during the decision phase of the agent.

The «interaction» stereotype tags tools that enable an agent to communicate directly or not with others or with its environment. Methods express the ability an agent owns to interact with others. Attributes represent functioning data or parameters of an interaction. Interactions can be classified into two groups: perceptions and actions which are also tagged with stereotypes («perception» and «action»). A coherency rule associated is: a method stereotyped with «interaction» can only call methods stereotyped with «skill» or «interaction». Moreover, all the methods appearing on protocol diagrams are automatically stereotyped «interaction».

The «cooperation» stereotype expresses that the social attitude of an agent is implemented using rules allowing Non Cooperative Situations (NCS) solving. An agent must have a set of rules (predicates) that enable it to detect NCS. It must also have a method to enable it to solve NCS, this method associates actions with situations in order to process them. A method that is stereotyped with «cooperation» is always called during the decision phase of an agent and can be of two kinds:

- A method that returns a Boolean result and tries to detect a NCS; its parameters are stereotyped with «perception», «representation» or «skill»,
- A solving method (a priori, one per agent) that allows the association of one or several solving actions with each NCS.

An associated coherency rule is: “a method stereotyped with «cooperation» is private.”

Define Non Cooperative Situations. This step represents another contribution of ADELFE to the design workflow. Rules which allow the agent to have a cooperative attitude have to be defined: how to detect and to remove NCS in order to be more cooperative. During it, designers must fill up a table describing each NCS encountered by each previously identified agent. This table contains:

- The name of the NCS.
- The state in which the agent is during the detection of NCS. This state can be defined by a set of values of attributes or results of methods which can be stereotyped as «perception», «Characteristic» or «representation».
- The textual description of the NCS.
- The conditions describe the different elements that enable to locally detect the NCS. Methods and attributes used to express conditions must be stereotyped as «perception» or «representation» or «skill».
- The actions linked to the NCS. The actions describe what the agent has to do to remove this NCS. Methods and attributes used to express actions must be stereotyped «action».

For each table, at least one «cooperation»-stereotyped method must be defined. This method corresponds to the NCS detection and will be expressed using the state and the conditions, i.e., methods and attributes that are stereotyped as «perception», «representation» or «Characteristic».

If several actions are possible to remove the detected NCS, you must define another method to choose the action to do. This method is stereotyped as «cooperation». If only one action is possible the definition of this second method is useless: this action will be always executed. These methods will be integrated in the behavior of the agent.

For instance, the NCS for a *BookingAgent* are:

- *Partnership incompetence*: the BA meets another BA that may be an uninteresting partner;
- *Booking incompetence*: the BA is in a cell that is uninteresting to book;
- *Message unproductiveness*: the BA receives a message that is not correctly addressed;
- *Partnership conflict*: the BA meets another BA that is interesting, but this other BA has already a partner;
- *Booking conflict*: the BA is in a cell that is interesting to book but this cell is already booked; and
- *Booking uselessness*: the BA meets its partner: they must separate to explore more efficiently the grid.

7. ADELFE Tools

Within ADELFE, three tools are associated with the process and the UML / AUML notations. The first tool is based on the OpenTool commercial software, enriched to take into account the development of adaptive MAS. The second tool is an interactive tool which supports the process and helps designers to follow the process and to execute associated tasks. The last tool is a support decision tool to help designers to decide if the AMAS theory is relevant for the current system to design. In this section we only present the two first tools.

7.1 OpenTool for ADELFE

OpenTool is a development tool, written in the OTScript language, which is designed and distributed by TNI-Valiosys, one of the ADELFE partners. On the one hand, OpenTool is a commercialized graphical tool like Rational Rose and supports the UML notation to model applications while assuring that the produced models are valid. More specifically, it focuses on analysis and design of software written in Java. On the other hand, OpenTool enables meta-modeling in order to design specific configurations. This latter feature has been used to extend OpenTool to take into account the specificities of adaptive MAS and thus include them into ADELFE.

The first modification added to OpenTool concerns the static view of the model: the class diagram. Nine stereotypes are integrated to modify the semantics of classes and features depending on the specificities of cooperative agents.

As ADELFE reuses AUML to model interactions, OpenTool was enhanced to support AIP diagrams. AIP diagrams are an extension to existing UML sequence diagrams that enables different message sending cardinalities (AND, OR or XOR). This second modification was enriched with the possibility to easily attach a protocol to an agent class. Moreover, in order to simulate agents' behaviors by using finite state machines, OpenTool can automatically generate state-chart diagrams corresponding to protocols and roles within these protocols.

7.2 Interactive Tools

ADELFE also provides an interactive tool that helps designers when following the process established in the method (Bernon et al., 2003). In classical object-oriented or agent-oriented methods, this kind of tool does not really exist. Even if some tools linked with agent-oriented methods exist, e.g., agentTool (DeLoach and Wood, 2001) (see chapter 6) for MaSE, PTK for PASSI (Cossentino, 2001) or the INGENIAS tool (Gomez-Sanz and Fuentes, 2002), they are not really a guide and a verification tool for designers follow-

ing a methodological process. Generally, some guides like books or HTML texts are given (e.g., a guide to follow the RUP is available on the Web site of Rational Software) but they are not really interactive tools able to follow a project through the different activities of the process. The ADELFE interactive tool is linked both with a tool to verify the AMAS adequacy and with OpenTool. It can communicate with OpenTool in order to access to different diagrams as process progresses. For these two reasons, it can be considered as a real guide that supports the notation adopted by the process and verifies the project consistency.

Each activity or step of the process is described by this tool and exemplified by applying it to the ETTO problem. Within the textual description or the example, some AMAS theory specific terms can be attached to a glossary in order to be explained. That is why the interactive tool is composed of several interfaces. The “Manager” interface indicates for the different opened projects, the different activities and steps designers have to follow when applying the methodology. The “Work Product” interface lists the work products that have been produced or that have to be produced yet regarding the progress when applying the methodology. The “Description” interface explains the different stages (activities or steps) designers must follow to apply the methodology process. The “Example” interface shows how the current stage has been applied to ETTO. The optional “Synthesis” interface shows a global view and an abstract of the already made activities. And finally the optional “Glossary” interface explains the terms used in the methodology and defines the stereotypes that have been added to UML.

8. Comparison with other Methodologies

ADELFE is based on object-oriented methodologies, follows the RUP (Rational Unified Process) and uses UML / AUML notations as MESSAGE (Caire et al., 2001 b) (see chapter 9). It covers the entire process of software engineering like MESSAGE, PASSI and Tropos (Castro et al., 2001) (see chapter 5). And as DESIRE (Brazier et al., 2000), MASSIVE, PASSI, Prometheus (see chapter 11), INGENIAS/MESSAGE (Gomez-Sanz and Fuentes, 2002), MaSE (DeLoach and Wood, 2001) (see chapter 6), it provides modeling graphical notations which are supported by tools. ADELFE is not a general-purpose method such as Gaia (Wooldridge et al., 1999) (see chapter 4) but it has a niche, which concerns applications that require adaptive MAS design using the AMAS theory. Therefore, like MESSAGE dedicated to telecoms applications, ADELFE gives guidelines for the identification of the application areas for which adaptive systems technology is better suited than other technologies, e.g., object-oriented technologies.

Many other methods, like AAIL (Kinny and Georgeff, 1996), Tropos, MaSE or MESSAGE, do not focus on the dynamic aspect of the software environment and on the adaptation abilities of the software. Tropos, like ADELFE, is concerned by dynamics. It expresses the dynamics and openness of the application in the requirements phases with the model of the environment and with particular soft goals. However, it does not give guidelines to design the right agents' behavior allowing the adaptability of the system.

In adaptive MAS, the environment (in which the system is operating) is a key notion; but in a general way, the environment modeling is not a central point in existing methodologies. In DESIRE, the environment is taken into account at the agent level in the *world interaction management module*: an agent maintains and interacts with its environment in the same way as with other agents. In Tropos, the environment model is described in terms of actors, their goals and interdependencies. In MESSAGE, the domain model captures some entities of the system environment and the interactions with the environment are described for each role in terms of sensory inputs and acquaintances, resources ownership and accesses, and finally tasks and actions. In AAIL, the relation between the agent and the environment is taken into account in the interaction model.

At the design level, some methodologies are dedicated to an agent architecture as AAIL with BDI, ADELFE with cooperative agents. In other methodologies such as Gaia, MESSAGE, Tropos, the architecture of the implemented agents is not defined and it is quite open. Note that Tropos offers different architecture styles (flat structure, pyramid, etc.) for its architectural design phase.

In the analysis workflow of Gaia, the agents are already identified and the methodology provides nothing to realize this identification. In Tropos the agents are found inside the actors' set. In AAIL, the elaboration and refinement of the agent model and the interaction model help the designer to define agents. The agent definition, which is given in MESSAGE and in ADELFE, defines the features that will be ascribed to the entities that the developer will choose to consider as agents.

9. Conclusion

The aim of this chapter was to present the ADELFE methodology which is a multiagent-oriented methodology suited to adaptive MAS based on the AMAS theory. ADELFE provides a new methodology to design a society of agents exhibiting a coherent activity. The first prototype is now operational and can be tested on the site <http://www.irit.fr/ADELFE>. Until now ADELFE has been used or is used in several case studies: an Intranet system design, a

timetabling problem, a flood forecast system (in progress), a mechanical design system (in progress) and a bioinformatics system (in progress).

Acknowledgments

We would like to thank the support of the French Ministry of Economy, Finances and Industry as well as our partners: TNI-Valiosys Ltd., ARTAL Technologies Ltd., the IRIT software engineering team, Carole Bernon and Valérie Camps.

Chapter 9

THE MESSAGE METHODOLOGY

Giovanni Caire, Wim Coulier, Francisco Garijo, Jorge Gómez-Sanz, Juan Pavón, Paul Kearney and Philippe Massonet

Abstract This chapter presents MESSAGE, an innovative agent oriented software engineering methodology, and it illustrates this methodology on an analysis and design case study. The methodology covers all phases of the software lifecycle, but focuses on MAS analysis and high-level design. It is intended for use in mainstream software engineering departments. MESSAGE integrates into a coherent AOSE methodology some basic agent related concepts such as organization, role, goal and task, that have so far been studied in isolation. The MESSAGE notation extends the UML with agent knowledge level concepts, and provides graphical notations for viewing them. The proposed diagrams extend UML class and activity diagrams.

1. Introduction

Agent technology is one of the main topics in the area of modern telecommunication applications. But without a reasonable engineering methodology for the development of agent based systems, professional development of these applications is informal, cumbersome, error prone and thus expensive. MESSAGE is an AOSE methodology designed for use in mainstream software engineering departments that develop complex distributed applications. MESSAGE consists of applicability guidelines, a modelling notation, and a process for analysis and design of agent systems. It is a genuinely agent-oriented methodology, but also builds upon the achievements of software engineering, and is consistent with current best practices. MESSAGE attempts to unify the best features of existing AOSE methodologies while grounding agent-oriented concepts in the same underlying semantic framework that UML (OMG, 2000c) uses, the standard modelling language in object oriented software engineering. It extends the UML Class and Activity diagrams with agent oriented concepts such as organization (Zambonelli et al., 2000), role (Kendall, 1998), goal (Dardenne et al., 1993) and task (Omicini, 2001). It also takes the Rational Unified

Process (RUP) (Kruchten, 2000) as software development process, and defines activities for identification and specification of MAS components in analysis and design.

Work towards an AOSE methodology can be divided into two broad categories. The first category aims to apply existing software engineering methodologies to AOSE. Agent UML (AUML) (see chapter 12), for example, defines extensions to UML with notations suited for agent concepts. AUML has extended UML's interaction diagrams to handle agent interaction protocols. The second category of work aims at developing a methodology from agent theory, mainly covering analysis and design. Typically these methodologies define a number of models for both analysis and design (Iglesias et al., 1998a), such as Gaia (Wooldridge et al., 2000b) (see chapter 4) and MAS-CommonKADS (Iglesias et al., 1998b). The Gaia methodology has two analysis models and three design models. MAS-CommonKADS has six models for analysis, and three for design. While these models are comprehensive, the process lacks a unifying semantic framework and notation. In addition to this work, goal modelling techniques have been useful for structuring the rationale behind a model (Dardenne et al., 1993).

This chapter presents an overview of MESSAGE. It illustrates its concepts and development process with a case study going through the phases of analysis, high-level and low-level design. Section 2 describes the MESSAGE modelling language and process. Section 3 describes an analysis and high-level design case study. Section 4 presents some design and implementation considerations based on experimentation with MESSAGE. Section 5 contains some reflections of the authors of MESSAGE about the methodology.

2. The MESSAGE Methodology

The MESSAGE methodology provides a graphical modelling language, a development process, and guidelines on how to apply the methodology, that cover at least the analysis and design phases. The methodology also explains the relationship to implementation, testing and deployment phases (Caire et al., 2001a).

2.1 The MESSAGE Modelling Language

MESSAGE is defined by means of the meta-modelling technique based on MOF that is used to specify UML (OMG, 2000c). The MESSAGE modelling language extends the basic UML concepts of Class and Association with knowledge level agent centric concepts (Caire et al., 2001b). This section presents the most important agent related concepts. They will be illustrated in section 3 in a case-study using graphical notations. The complete

language definition is available at the official MESSAGE Web site <http://www.eurescom.de/Public/Projects/p900-series/P907/P907.htm>.

Agent. An Agent is an atomic autonomous entity that is capable of performing some (potentially) useful function. The functional capability is captured as the agent's services. A *service* is the knowledge level analogue of an object's operation. The quality of autonomy means that an agent's *actions* are not solely dictated by external events or interactions, but also by its own motivation. This motivation is captured as an attribute named *purpose*. The purpose will, for example, influence whether an agent agrees to a request to perform a service and also the way it provides the service.

Organization. An Organization is a group of Agents working together to a common purpose. Its services are provided collectively by its constituent agents. It has structure expressed through *power relationships* (e.g., superior-subordinate relationships) between constituents, and behaviour/co-ordination mechanisms expressed through Interactions between constituents.

Role. The distinction between Role and Agent is analogous to that between Interface and (object) Class: a Role describes the external characteristics of an Agent in a particular context. An Agent may be capable of playing several roles, and multiple Agents may be able to play the same Role. Roles can also be used as indirect references to Agents. This is useful in defining re-usable patterns.

Goal. A Goal associates an Agent with a state. If a Goal instance is present in the Agent's working memory, then the Agent intends to bring about the state referenced by the Goal.

Task. To express agent actions, MESSAGE uses activities. The main types of activity are Task and Interaction. A Task is a knowledge-level unit of activity with a single prime performer. A task has a set of pairs of Situations describing pre- and post-conditions. If the Task is performed when a pre-condition is valid, then one can expect the associated post-condition to hold when the Task is completed. Composite Tasks can be expressed in terms of causally linked sub-tasks (which may have different performers from the parent Task). Tasks are StateMachines, so that, e.g., UML activity diagrams can be used to show temporal dependencies of sub-tasks.

Interaction and InteractionProtocol. The MESSAGE concept of Interaction is another type of activity and borrows heavily from the Gaia methodology (see chapter 4). An Interaction by definition has more than one participant, and

a purpose which the participants collectively must aim to achieve. The purpose typically is to reach a consistent view of some aspect of the problem domain, to agree terms of a service or to exchange to results of one or more services. An InteractionProtocol defines a pattern of message exchange associated with an Interaction.

2.2 MESSAGE Analysis Model and Views

A MESSAGE analysis model is a complex network of inter-related classes and instances derived from concepts defined in the MESSAGE meta-model. MESSAGE defines a number of views that focus on overlapping sub-sets of entity and relationship concepts.

Five views have been defined to help the modeler focus on coherent subsets of the modelling language:

- Organization;
- Goal/Task;
- Agent/Role;
- Interaction; and
- Domain.

The Organization view shows concrete entities, i.e., Agents, Organizations, Roles, Resources (such as databases and application services), in the system and its environment and coarse-grained relationships between them (aggregation, power, and acquaintance relationships). The Goal/Task view shows a detail of the goals that the Agents/Roles pursue and the tasks that they perform to reach them. The Agent/Role view focuses on the individual agents and roles. This view describes their characteristics, such as what goals they are responsible for, what events they need to sense, what resources they control, the behavior rules needed, etc. The Interaction view shows, for each interaction among Agents/Roles, the initiator, the collaborators, the motivator (generally a goal the initiator is responsible for), the relevant information supplied/achieved by each participant, the events that trigger the interaction, and other relevant effects of the interaction (e.g., an agent becoming responsible for a new goal). Finally, the Domain view shows the domain specific concepts and relations that are relevant for the system under development.

These five views may be graphically visualized through new diagram types: Organization, Goal/Task, Delegation and Interaction, which extend the UML Class diagram, and Workflow which extends the UML Activity diagram. They are illustrated in the case-study below.

2.3 MESSAGE Design Model

The MESSAGE Design Model refines the analysis model. It provides detailed agent interaction constructs to describe inter-agent communication and information exchange between agents and with their environment. The design model also provides means (such as a facilitator or directory agent) whereby the agent can identify other agents with which to communicate. The design model also models the agent organization. This covers the capability of agents to co-operate with other agents for problem solving. The Agent view of the Design Model also describes the agent's internal structure and behavior. The internal design of the individual agents is described in terms of reusable components. Constructs for describing basic agent concepts are defined: observation, action, communication, goals, plans, etc. These design concepts are related to corresponding concepts in the analysis model, but address issues about how these concepts should be implemented. MAS design also takes into account platform choices and adherence to standards such as FIPA. Although in principle high-level design should be independent of a specific platform, design abstractions constrain designs to a specific family of platforms. The MESSAGE design process is based on the selection for each agent of an agent architecture, the refinement of the analysis model into a design model, and the allocation of the model elements to the agent architecture (Caire et al., 2001b).

2.4 The Analysis and Design Process

MESSAGE has adopted the Rational Unified Process (RUP) (Kruchten, 2000) as its generic software engineering project life-cycle framework. MESSAGE focuses on the analysis and design activities. Both of these are modelling activities: the main output of each is a model of the system at an appropriate level of abstraction.

The purpose of analysis is to produce a system specification (or analysis model) that interprets the problem to be solved (i.e., the requirements) represented as an abstract model in order to (i) understand the problem better; (ii) confirm that this is the right problem to solve (validation); and (iii) facilitate design of the solution. It must therefore be related both to the statement of requirements and to the design model (which is an abstract description of the solution). MAS analysis focuses on defining the domain of discourse and describing the organizations involved in the MAS, their goals and the roles they have defined to satisfy them. The high-level goals are decomposed and satisfied in terms of services provided by roles. The interactions between roles that are needed to satisfy the goals are also described. The analysis models are produced by stepwise refinement.

In design, MESSAGE distinguishes between high-level design, which is implementation independent, and low-level design that takes into account the

specific constraints of a target agent platform such as the agent architecture and the knowledge representations. In high-level design the analysis model is refined by assigning roles to agents and by describing how the services are provided in terms of tasks. The tasks can be decomposed into direct actions on the agent's internal representation of the environment, and communicative actions to send and receive messages in interaction protocols. The interactions between roles identified in analysis are detailed using interaction protocols.

The low-level design assumes that the developer is thinking about possible implementations. This process implies considering different mappings from high-level design concepts to computational elements provided by the target development platforms. By computational we mean having an application program interface with an externally known behavior. These elements may already exist, e.g., as a software library, or will need to be developed from scratch. Examples of both approaches will be shown later. Implementation from the low-level design is not different from the implementation stage in a common software development, so it will not be considered further.

In each stage, the developer needs to perform stepwise refinement of the model. MESSAGE has defined for analysis some refinement strategies. To structure the refinement, MESSAGE proposes levels of refinement. Different levels are numbered starting with level 0. Each level starts with a set of elements which are modified using different refinement strategies. A level, then, contains information about the system with an abstraction degree inversely proportional to the number of the level.

Refinement Approach. Level 0 is concerned with defining the system to be developed with respect to its stakeholders and environment. The system appears as a set of organizations that interact with resources, actors, or other organizations. Actors may be human users or other existing agents. Subsequent stages of refinement result in the creation of models at level 1, level 2 and so on.

At level 0 the modelling process starts building the Organization and the Goal/Task views. These views then act as inputs to creating the Agent/Role and the Domain Views. Finally the Interaction view is built using input from the other models. The level 0 model gives an overall view of the system, its environment, and its global functionality. The granularity of level 0 focuses on the identification of entities and their relationships according to the meta-model. The following levels determine the structure and the behavior of entities such as organization, agents, tasks, goals and domain entities. Additional levels might be defined for analyzing specific aspects of the system dealing with functional requirements and non functional requirements such as performance, distribution, fault tolerance, security. Since each level expands concepts appearing in previous levels, it is recommended to execute consistency

checks periodically. In the following case-study only level 0 and level 1 are illustrated.

3. Analysis/Design Travel Agent Case-Study

This section presents a case study developed to evaluate the methodology. It contains diagrams extracted from internal documentation of the project. Partial implementations of this case study are available from the official MESSAGE Web site <http://www.eurescom.de/Public/Projects/p900-series/P907/P907.htm>.

3.1 Travel Agent Case-Study Description

Context. A travel service provider (TSP) wants to improve its services by helping its clients to obtain complete travel plans, including taxis, restaurants, and so on. Travelling from one location to another involves creating a travel plan with a tight schedule. It might involve taking a taxi from one's home to the airport, taking a flight to an intermediate location, taking a connection to the final destination where a rented car has been booked and can be picked up to drive to the hotel where reservations have been made. Unfortunately for the traveller, many things can go wrong with a travel plan.

Requirements. Given the fact that many travellers will soon have wireless terminals, the efficiency of the travelling process can be improved by developing a system (distributed both on these terminals and on the terrestrial network) that:

- Gathers travel requirements from the traveller;
- Assists the traveller in identifying and arranging relevant travel services offered by the travel service providers;
- Assists the traveller with the booking of travel tickets; and
- Monitors that the travel arrangement is carried out as planned by providing alerts and notifications of changes to arranged travels, and proposing alternatives for unexpected changes in the schedule.

3.2 Level 0 Analysis

Organizations. Two diagrams from the level 0 organization view are reported. Figure 9.1 describes structural relationships in a level 0 organization diagram. The system to design is a Knowledge Management (KM) system, which is considered a part of the TSP organization. The TSP already contains some existing infrastructure, staff, and workflows. At level 0 the system under

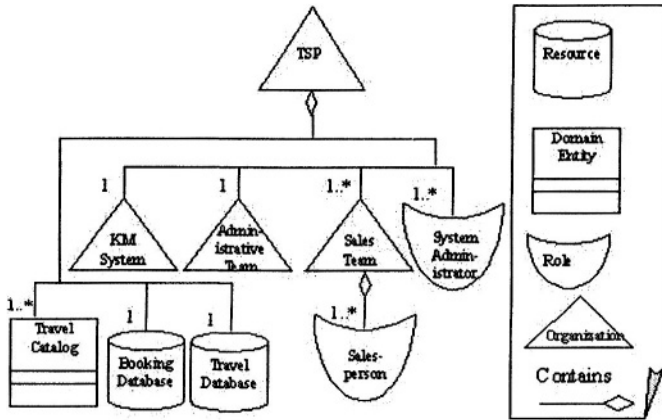


Figure 9.1. Level 0 Organization diagram (structural relationships)

development, i.e., the KM System, is seen itself as an organization that will be analyzed at level 1. At this level of modelling only the global organizations are modelled, not the individual software agents.

Figure 9.2 shows the acquaintance relationships in the level 0 organization diagram. The KM system interacts with the System Administrator and the Salesperson, and with the Travel Database to retrieve Travel Arrangements and the Booking Database to insert the bookings requested by Salesperson on behalf of Travellers. Moreover, it interacts with the Administrative Team to prepare the bills that will be sent to travellers. A Salesperson interacts with Travellers to gather Travel Requirements and provide Travel Arrangements. It has to be noticed that the Salesperson does not interact directly with the Travel Database and the Booking Database. All these interactions are carried out through the KM system.

Goals/tasks. The Goal view shows how the main goals of the system are structured into sub-goals (Dardenne et al., 1993). The level 0 goal view shows the common goals that are pursued by the organizations that interact, as well as their own. For example, Figure 9.5 shows that the main goal of the system (Traveller Assisted) is satisfied when a Travel Arrangement (TA) for the Travelling Requirements (TR) has been selected, and that the Traveller is notified of problems that will prevent the Travel Arrangement from being followed. The TravelArrangementProvided goal is satisfied when the TR are known, TAs have been provided by TSPs, the TA have been ranked, and a TA has been selected by the Traveller. The decomposition of the NotifiedOfTravelProblems goal is not shown.

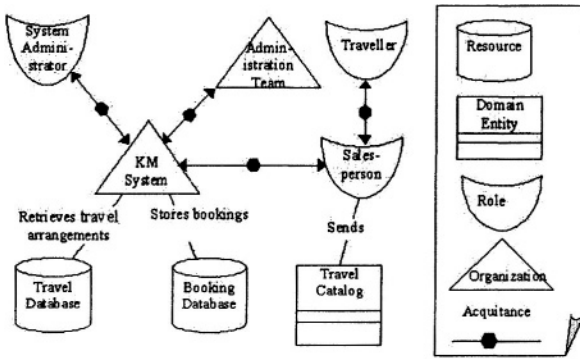


Figure 9.2. Level 0 Organization diagram (acquittance relationships)

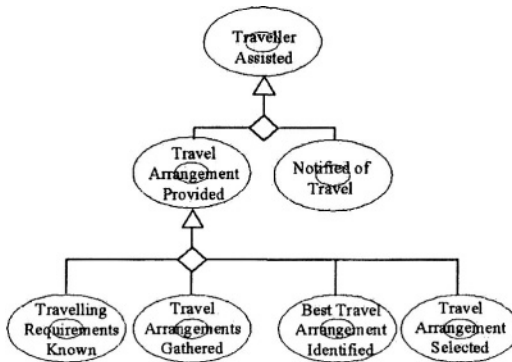


Figure 9.3. Level 0 Goal/Task Implication diagram

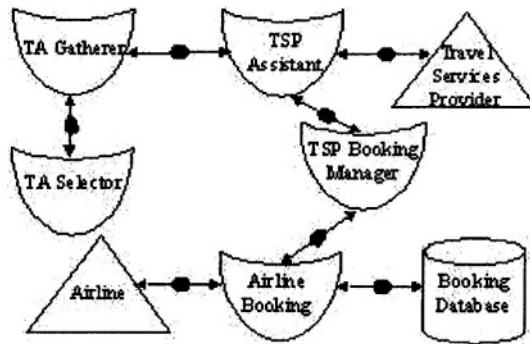


Figure 9.4. Organizations, Roles and their interactions

Alternatively, or in conjunction with goal decomposition it is useful to analyze how a given service is realized by a partially ordered set of tasks. An example for level 1 is shown in Figure 9.9. The descriptions of workflows can be refined as analysis/design progresses.

3.3 Level 1 Analysis

In this level, analysis focuses on the system itself identifying the main functionality required (seen as roles and/or particular types of agents). The approach followed in this simple case study is to only consider roles during analysis and to defer the identification of agents and what roles they will play to the beginning of the design process. However, the developer is free to start identifying agents during analysis.

Organizations. In the final system, a Traveller, i.e., the user, will have to interact with two roles: TA Gatherer which is responsible for gathering TAs for a given TR, and TA Selector which is responsible for ranking the TAs that best match the TR. Figure 9.4 shows these roles together with two other kind of organizations which are already involved: the TSP and the Airline Company (AC). The TA Gatherer role interacts with the TSP Assistant role of the TSP to gather the TA. When the Traveller requests a Booking, the TSP Booking Manager role interacts with the Airline Booking role of the airline company to make the reservation in the Booking database, which is modelled as a resource. These interactions will be described later in the high-level design section using interaction protocols.

Goals, Roles and Services. Organizations have high-level goals that the roles need to satisfy by providing and requesting services. Usually, the orga-

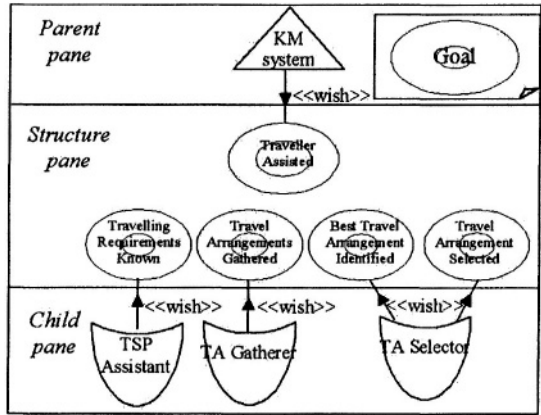


Figure 9.5. Level 1 Delegation Structure diagram

nization goals are decomposed into simpler goals. For example, the Traveller Assisted goal of the Traveller is partially satisfied by the goal TAGathered of the TAGatherer role, and the goals BestTAIdentified and TASelected of the TASelector role. These roles can satisfy those goals because they provide the services TAGathering and BestTASelection. The roles responsible for satisfying the different goals appear in Figure 9.5. This information complements that shown in workflow diagrams illustrated in Figure 9.9.

Figure 9.5 shows a delegation structure diagram. Only the root and the leaves of the decomposition of the parent organization goal are shown. This diagram is strictly related to (and must be consistent with) both the goal decomposition diagram showing the decomposition of the organization goal and the organization diagram showing the Agents/Roles inside the organization. Information about what task execution to provide a service appear in workflow diagrams. At the analysis level, this information is typically quite informal and therefore free text is preferred to a graphical notation.

Interactions. This view highlights which, why, and when, Agents/Roles need to communicate leaving all the details about how the communication takes place to the design process. The interaction view is typically refined through several iterations as long as new interactions are discovered. It can be expressed by means of a number of interaction diagrams. Figure 9.6 shows an interaction diagram describing the Travelling Request interaction between the TAGatherer and the TSPAssistant roles. The details of the interaction protocol and the messages that are exchanged between roles can be represented using Auml sequence diagram (see chapter 12) or any other notation that is considered convenient.

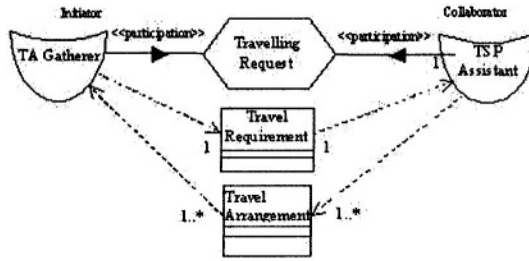


Figure 9.6. Interaction diagram

Domain. The domain can be conveniently described using UML class diagrams to model the classes, their associations and their attributes, as shown in Figure 9.7. A TR consists of a set of Transfer Requirements (TsR) that specify an origin, a destination, and a time frame. A TA is composed of a set of Transfer Arrangement (TsA) that refers to a Flight Occurrence (FO). A FO refers to a flight from an origin to a destination on a specific date. A TA matches a TR if each TsR is matched by a TsA.

3.4 Transition to a High-Level Design

This section shows how to refine an analysis model into a high-level design. Four steps are proposed: identifying agents and assigning them roles, describing how services are provided with tasks, refining the analysis interactions in terms of interaction protocols, and specifying the behavior of the interaction protocols roles as statecharts.

Assigning Roles to Agents. Agents are identified based on the description of the organizations and the use of some heuristics: on one extreme there can be one agent for each organization, and is assigned all the roles in the multiagent application. On the other extreme, there is one agent per role. In this case study a Personal Travel Agent (PTA) was created for the Traveller, a TSP agent was created for the TSP organization, and an Airline agent was created for the AC organization. Figure 9.8 shows that for playing these two roles, the TSP agent must provide the ProvideTAs service, manage the TR database, and access the FlightDestinations database. It also uses services from the PTA, and the service ProvideFlightAvailability.

Providing Service with Tasks. If any workflow diagrams have been defined in the analysis phase they can be further refined in this stage. Once the agents have been assigned roles, the services they need to provide are known. For each agent it is necessary to show how the services will be provided in

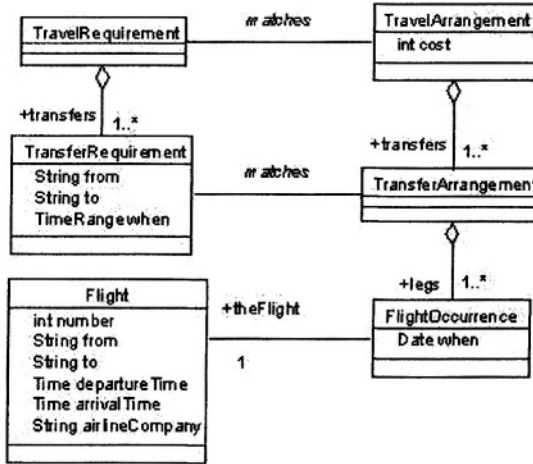


Figure 9.7. Travel domain

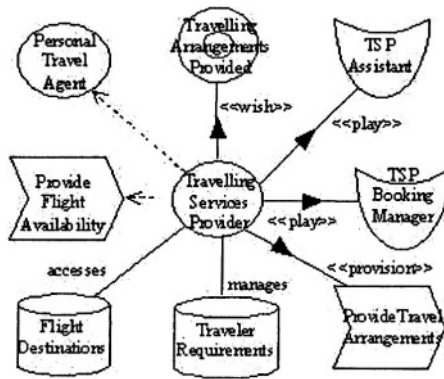


Figure 9.8. Agent diagram

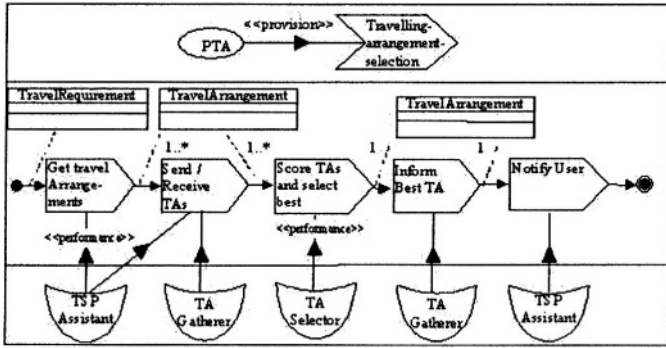


Figure 9.9. Task workflow

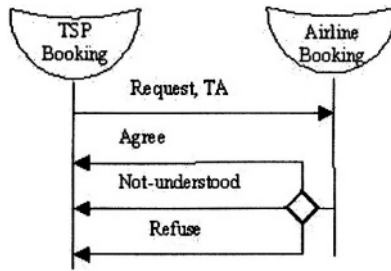


Figure 9.10. Partial Request Interaction protocol

terms of the tasks it is capable of. Figure 9.9 shows the workflow of tasks that is needed for the PTA agent to provide the TA Selection service to the Traveller. The input to the GetTAs task of the TSP Assistant role is a TR using the object flow UML notation. The output is a set of TA that is sent to the TA Gatherer role, which then passes them to the TA Selector role to rank them.

Interaction Protocols. This step involves refining the interactions identified in analysis. Interactions can be modelled in terms of interaction protocols and UML state-charts. This modelling takes into account the interactions between roles, the assignment of agents to roles, and the implementation of services in terms of tasks, direct actions and communicative actions. Figure 9.10 shows how the interaction between the TSP BookingManager and the AirlineBooking roles can be modelled as a FIPA-Request interaction protocol. A TravelArrangement is passed as the content of the request message.

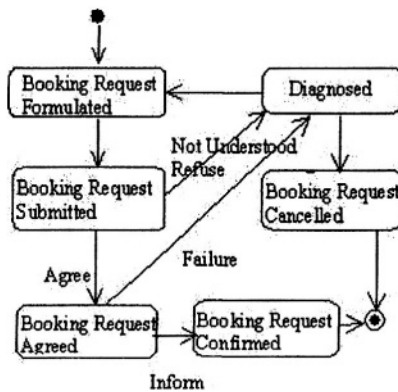


Figure 9.11. TSP Booking Manager Role state chart

Interaction Role Behavior Specification. This step involves modelling the behavior of the roles in an interaction protocol. Figure 9.11 shows how the behavior of TSP Booking Manager role can be modelled during the FIPA-Request protocol. In this case, when a booking request is refused or not understood, it is diagnosed and a decision is made to either cancel the request or retry the request.

4. Considerations on Low-Level Design

Given a high-level MAS design model, low-level design needs to define computational entities that can be implemented. This means translating the entities in terms of subsystems, interfaces, classes, operation signatures, algorithms, objects, and other computational concepts. Since there is no enough space in the chapter to consider an in depth translation procedure, this section will summarize the main results in this respect. The MESSAGE has experimented with two complementary design approaches (Caire et al., 2001a).

The first design approach is driven by the MAS organization and an agent architecture. It considers the agent as an entity that is more than a class: an agent is seen as a subsystem, with an internal architecture that defines the relationships of the different agent components. These components are computational entities that are identified and built by transformation and stepwise refinement from the analysis models. The behavior of some of these components (e.g., agent control) can be complex (e.g., defined using a BDI approach) or simple (e.g., in reactive agents). The design process is driven by the organization model in order to assign responsibilities, to define agent interactions, and to model social knowledge. The detailed design decisions that were applied in

one of the case studies of MESSAGE have been presented in (Garijo et al, 2001).

The second approach is more agent-platform oriented, and considers that each agent can be mapped to a class. This is mainly derived from the application of most agent models supported by agent building tools, such as JADE (Bellifemine et al., 2001) (see chapter 13) or FIPA-OS (Poslad et al., 2000) in which there is one Agent class from which to derive the specific agent type. This approach is valid when agents are simple in behavior and can be modelled using a classical state machine approach, e.g., as in typical object-oriented languages such as Java. The main concern here is how to organize agent interactions. An example of such implementation was published in (Massonet et al., 2002).

4.1 Organization Driven Design

The Organization model defines the architectural framework for achieving design activities. It provides a high level view of the system with all the elements needed for structuring their computational entities. Studying the organization, a developer discovers what are the goals to be satisfied and why an agent decides to collaborate with others.

The organization itself may or may not be present as a computational element at the end of the design. An organization may be designed with an agent architecture, or distributed among agents in form of social knowledge (abilities of different roles and expected behavior), or appear in form of shared resources, like resources to manage the list of members in a organization. Interfaces, tasks, and agent architectures are selected while studying organization requirements. The organization view shows which roles need to interact, their communication needs, and what social knowledge is required from each agent in order to satisfy the system goals, i.e., organization goals. Individual realization of tasks or service provision is easier to design as collective realization. Once the organization has been modelled the developer can proceed with:

- Selecting an agent architecture. Agents are designed as instantiations of a specific kind of agent architecture, whose complexity depends on the roles that have been assigned to the agents in the organization, and the kind of relationships with other agents (e.g., whether interactions involve complex protocols or not). In MESSAGE there have been experiments with cognitive (BDI agents) and reactive architectures (state-machine based agents).
- Specifying agent's behavior and interfaces. Conventional Software Engineering modelling techniques can help to detail internal agent architecture behavior. For instance, sequence diagrams to clarify interactions, activity diagrams to model the sequence of states reached by agents

when performing tasks, and use cases to detail the expected functionality of the final system.

- Defining agent's social knowledge: this is defined using the structure and the relationships of the Organization Model. It supports reasoning about other agent's actions, about the society itself, and social constraints upon agent's actions.

4.2 FIPA Platform Dependent Design/Implementation

A case study on the transition between a MESSAGE specification and FIPA compliant agent toolkit was carried out with the JADE framework and the JESS (Friedman-Hill, 2003) rule based system for reasoning (Massonet et al., 2002). It focused on the translation between the implementation independent MESSAGE analysis/design concepts and the implementation dependent JADE detailed design/implementation concepts. It was illustrated on a subset of the case study described in section 3, and showed how the high-level agent concepts of the analysis and design modelling language could help structure the agent implementation that is usually based on a simpler set of agent concepts. High-level design decisions were easier to make with MESSAGE than at the implementation level. The case study also showed, that because the development tools have their own set of basic agent concepts, the translation process is unique for every combination of methodology and development tool.

5. Evaluation of MESSAGE

The MESSAGE methodology was evaluated on two case studies Analysis of a Universal Personal Assistant for Travel and an adaptive and decentralized Customer Service Operations Support System application performing end-to-end co-ordination within a customer service business process. A preliminary evaluation concluded that MESSAGE showed promise, but the following issues needed attention:

- A clearer and more consistent semantics was needed for the basic entity and relationship concepts in the modelling language, and particularly those used in the diagrams.
- A graphical notation distinguishing visually the fundamental MESSAGE entities and relationships from the default UML ones was needed, plus distinct diagram types focussing on particular aspects of the analysis model.
- A clearer process model and more specific guidelines were required to give practical assistance to the developer.

These issues were addressed in the final version of the methodology. The design process was defined in parallel with the development case study, so that the case study provided source material for the methodology rather than being a test of it. Consequently, no explicit evaluation of the design process has been performed. A subjective assessment based on reading the final methodology is that the design process is at a similar stage of maturity as the initial version of the analysis process. There are many useful ideas present derived from the experience of the project partners, but they need to be marshalled so as to tell a consistent and coherent story. Much of the design process represents a refinement of the corresponding elements of the analysis process. No additional notation is introduced: a mixture of UML diagrams, MESSAGE analysis notation and ad hoc notations are used. However, being able to perform a partial implementation from the design information should be considered.

6. Conclusions

This chapter has presented the MESSAGE AOSE methodology and illustrated it on an analysis and design case study. The MESSAGE notation extends UML by contributing agent knowledge level concepts, and diagrams for viewing them. The diagrams extend UML Class and Activity diagrams. The methodology covers MAS analysis and design and is designed for use in mainstream software engineering departments.

The case studies showed that the MESSAGE agent centered concepts proved to be a sufficiently complete set of construction primitives for the case study problems. Also, using the views of the system as building patterns helps developers obtain a more complete specification of the MAS. MESSAGE, as it stands, is not a complete, mature agent-oriented methodology. It does, however, make some significant practical contributions to the state of the art (Garijo et al, 2001) that are likely to influence on-going initiatives in this area, e.g., Agent UML (OMG, 2000c) (see chapter 12). In particular, the graphical notation and the diagram set are practical concrete results that could be taken up widely. These have been the basis for new proposals, such as INGENIAS (Gomez-Sanz and Pavon, 2003), which refines the different meta-models and adds more tool support, and RT-MESSAGE (Julian and Botti, 2004), with extensions to deal with real-time constraints.

Acknowledgments

This work was supported by EURESCOM project P907. The editing of this chapter was mainly supported by FEDER and the Walloon region under contract Convention EP1A1203000012 number DGTRE 130023, and by the Spanish Ministry for Science and Technology (TIC2002-04516-C03-03).

Chapter 10

THE SADDE METHODOLOGY

Social Agents Design Driven by Equations

Carles Sierra, Jordi Sabater, Jaume Agusti and Pere Garcia

Abstract This work explores the existing gap between multi-agent specification and implementation and the potential help that evolutionary programming techniques can bring in. We present a methodology to help the programmer in the transition from a set of desired global properties expressed as an equation-based model that a MAS must fulfill to an actual society of interacting agents. The evolutionary techniques are seen, within this methodology, as a procedure to tune the parameters of the population of agents in order that their aggregated behaviour maximally approaches the desired global properties.

1. Introduction

A fundamental difference between the ecologist and chemist and the software engineer is that lions, gazelles and atoms already exist. They are *natural*. Scientist do not need to *design* them. Their task consists on observing phenomena and building a set of equations for which the observed reality is a model. If the predictions of the equations and the reality do not match, the set of equations is wrong. The scientist then refines the equations until predictions and reality match. It works opposite to the methodological approach presented here, as we hope to make clear by the end of it.

The general goal of the research reported here is to better understand the dynamics of large (*artificial*) MAS with globally distributed and interconnected collections of human, software and hardware systems; each one of which with potentially thousands of components.

To understand these dynamics we take a different stance than the traditional *emergent behaviour* community. We focus our attention on the study of the relationships between the *a priori* desired global behaviour of an agent society and the actual emergent behaviour shown by the group of agents forming the society. In a sense, we feel that in order to have a handle into the engineering

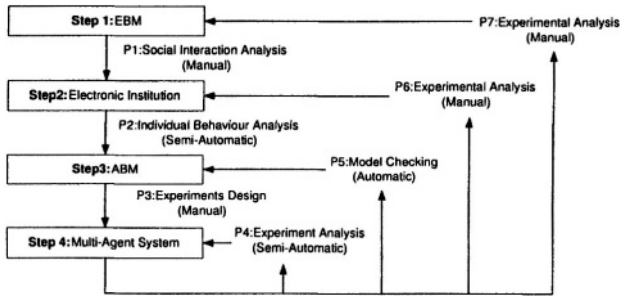


Figure 10.1. SADDE Methodology

of complex systems we have to first specify the desired behaviour and second find ways to restrict the potential complex interactions among agents in order to foresee an emergent behaviour that does not depart substantially from what is expected. Within this ambitious goal we present a methodology based on three main ideas. First, a particular approach to the principled design of MAS using Equation-Based Models (EBM, for short) as a high level specification method, where equations model the aggregated behaviour of the agent populations abstracting from the interaction details of individual agents. Second, the use of *Electronic Institutions*, as the way to restrict the interaction among agents in order to be able to *engineer* the emergence. Third, the use of evolutionary computation techniques to find out what agent structures produce the behaviour specified in the EBM. These ideas frame our design methodology called SADDE (Social Agents Design Driven by Equations).

2. The SADDE Methodology

We take the stance that in order to build a model for a society containing thousands of agents, the general view provided by an EBM provides succinct descriptions of population-level behaviours which we then attempt to replicate using models consisting of a society of individual interacting agents. Our proposed lifecycle is graphically depicted in Figure 10.1.

An important characteristic of MASs design from a software engineering perspective is the decoupling of the interaction process between agents from the deliberative/reactive activity in each agent (Ferber and Gutknecht, 1998). The notion of *electronic institution* (Noriega and Sierra, 1999; Rodríguez et al., 1998) plays this role in our methodology by establishing a framework that constraints and enforces the acceptable behaviour of agents.

The different phases within SADDE are:

Step 1 EBM – Equation-Based Model. In this first step, a set of state variables and equations relating them must be identified. These equations

have to model the desired global behaviour of the agent society and will not contain references to individuals of that society. Typically these variables will refer to values in the environment and to averages of predictions for observable variables of the agents. The EBM is the starting point of the construction of a system that later on will be observed. Thus, a comparison between the EBM predicted behaviour and the actual ABM behaviour will be obtained.

Step 2 EIM – Electronic Institution Model. In this step the interactions among agents are the focus. It is a first “zoom in” of the methodology from the global view towards the individual models. This step is not a refinement of the EBM but the design of a set of social interaction norms that are consistent with the relations established in Step 1. The EBM does not necessarily reflect by itself the set of agent roles that might generate the relations between the global variables. It is the task of the engineer to determine which roles will be present at the level of the society design by means of an electronic institution.

Electronic Institution restrict the interaction between agents in several ways: enforcing protocols (when an to whom say what), restricting movements of agents among activities (scenes) and by enforcing norms that restrict the actions of agents. This restrictions permit to engineer emergence to a certain level in the sense that agents are not completely free to act.

Step 3 ABM – Agent-Based Model. Here, we focus in the individual. We have to decide what decision models to use. This is the second “zoom in” of the methodology. New elements of the requirement analysis (new variables) will be taken into account here. For instance, some rationality principles associated to agents (e.g., producers do not sell below production costs), or negotiation models to be used, e.g., as those proposed in (Sierra et al., 1997), have to be selected.

Step 4 Multiagent System. Finally, the last step of our methodology consists on the design of experiments for the interaction of large numbers of agents designed in the previous step. For each type of agent the number of individuals and the concrete setting for the parameters will be the matter of decision here. The results of these experiments will determine whether the requirements of the artificial society so constructed have been consistently interpreted throughout the methodology and thus whether the expected results according to the EBM are confirmed or not.

Once the experiments designed at Step 4 are run and analysed, several re-designs are possible as shown schematically in Figure 10.1. The different forward and backward processes of the methodology are:

- P1 Social Interaction Analysis.** Once the EBM has been constructed, the relations between the global variables and the analysis of the requirements of the society to model will determine what sort of agents exist (i.e., the roles), what sort of interactions the agents must have (i.e., the scenes), and what sort of transactions or dialogs they will have (i.e., ontology). This is an inherently manual process: there are many decisions to be made at this stage that have not been specified in the EBM.
- P2 Individual Behaviour Analysis.** Once a complete picture of the institution is ready, the final aspect to consider is the modeling of the behaviour of the agents. Many aspects of this behaviour are already determined by the institution. For those aspects that are not completely determined the methodology strongly encourages the design of parametric decision models to fill in the gaps. These parameters will be used to set different experiments and will be the target of agent design rules.
- P3 Experiment Design.** By choosing agents to participate with (possibly) different decision mechanisms, and by giving concrete values to the parameters of those decision mechanisms, different experiments can be constructed. The experiments should be set so as to explore all the possibilities and to see whether the EBM is making the right prognosis.
- P4 Experiment Analysis (ABM redesign).** The analysis of the experiments will be done by comparing the predicted values of the global variables by the EBM and the actual values of agent variables and their averages.
- P5 Model Checking.** The claims about the behaviour of a group of agents that the developer establishes when specifying an experiment will be model-checked at this stage. The outcome of the model checking will help to change the agent-based models, i.e., change the decision-making models.
- P6 Experiment Analysis (EI redesign).** Additionally, when model checking determines that certain properties can never be guaranteed or that after several trials it is impossible to find parameter values that lead to the expected correct behaviour, different constraints over the agents interactions could be explored. This means that a redesign of the EI may be in place. This is an intrinsically manual task.
- P7 Experiment Analysis (EBM redesign).** Finally, and if everything fails, it may happen that the part of the requirements that led to the initial EBM was misunderstood and that a variation in the initial EBM is necessary to explain why the experiments are showing unexpected behaviours.

3. A Case Study: The Electricity Market

An electricity market is a special kind of market where participants trade with power. It has three main components: producers, consumers and a network that is responsible of distributing the power from producers to consumers. This network has physical restrictions that makes necessary the presence of an external entity, the system operator (SO), that tries to agree the offer and the demand while maintaining the network into the safety operation limits.

Due to this characteristic, an electricity market always has two stages. In the first stage, producers and consumers participate in one (or several) free markets (explicitly forbidden for the system operator) in order to exchange power. After this stage, there is another stage (that is performed just before the power traded in the first stage has to be introduced into the net) where the system operator analyzes the offer and the demand and detects possible problems for the net. If problems are identified, the system operator has several mechanisms to avoid or minimize them as much as possible.

In the following sections we will apply step by step the SADDE methodology to this scenario.

4. Step 1: The EBM

In the EBM we have modelled the three main components of the electricity market: generation, consumption and the electrical network system operator.

4.1 Power Generation

The power production has been modelled using three types of power stations: Thermic (coal-fired, gas-fired and fuel-fired), Nuclear and Hydroelectric. The features of these power stations have been taken from the existing ones in the Spanish electrical market during the year 2001. Hydraulic power stations generated a 21% of the global power, Nuclear power stations a 35% and Thermic power stations a 44%. Using these proportions as a reference we can compute the power of each type of power station taking into account that we want to obtain for our scenario a global power around 40000 MW. Concretely in our scenario 9000 MW are generated using hydraulic power stations, 14000 MW using nuclear power stations and 18000 MW using thermic power stations.

In the EBM, we will have a single entity that models the energy production for each type of power station: hydraulic, nuclear and thermic.

Decision Modules. Each power generation entity in the EBM (Hydraulic, Nuclear and Thermic) uses a decision module to control the increase or decrease of power production from time t to time $t + 1$ taking into account the following criteria:

- The changes of power demand between time $t - 2$, $t - 1$ and t using the following function:

$$x = \begin{cases} \text{var2} + \text{var4} & \text{IF } \text{var3} > 0 \text{ AND } \text{var2} > 0 \text{ AND } \text{var4} > 0 \\ (\text{var2} + \text{var3})/2 & \text{IF } \text{var2} \leq 0 \text{ AND } \text{var3} > 0 \\ \text{var3}/2 & \text{IF } \text{var2} \leq 0 \text{ AND } \text{var3} \leq 0 \text{ AND } \text{var4} < 0 \\ \text{var2} + \text{var4}/2 & \text{IF } \text{var2} \leq 0 \text{ AND } \text{var3} < 0 \text{ AND } \text{var4} > 0 \\ \text{var2} & \text{OTHERWISE} \end{cases}$$

where var2 is the increase or decrease of power consumption between $t - 1$ and t , var3 is the increase or decrease of power consumption between $t - 2$ and $t - 1$ and $\text{var4} = \text{var2} - \text{var3}$.

- The performance, $\text{EnergyProd}(t)/\text{MaxN}$, of a power generation entity at time t in comparison to $\text{TotalDemand}(t)/41000$, the global performance of the system with a performance limit of a 70%

$$y = \begin{cases} \max\left(x, \frac{\text{MaxN} \cdot \text{TotalDemand}(t)}{41000 - \text{EnergyProd}(t)}\right) & \text{IF } \left(\frac{\text{TotalDemand}(t)}{41000} > \frac{\text{EnergyProd}(t)}{\text{MaxN}}\right) \\ & \text{AND } \text{excess}(t) = 0 \\ x & \text{IF } \frac{\text{EnergyProd}(t)}{\text{MaxN}} < 0.7 \\ 0 & \text{IF } x > 0 \text{ AND } \frac{\text{EnergyProd}(t)}{\text{MaxN}} \geq 0.7 \\ \min(x, 0.7 \cdot \text{MaxN} - \text{EnergyProd}(t)) & \text{OTHERWISE} \end{cases}$$

- Every power generation entity have to produce spear energy (i.e., $\text{Reserve}(t+1)$) to avoid possible power shortages.

$$z = \begin{cases} \max\left(\frac{\text{Reserve}(t+1)}{3} - \frac{\text{SpearEnergy}(t)}{3}, y - \frac{\text{SpearEnergy}(t)}{3}\right) & \text{IF } \text{SpearEnergy}(t) < 0 \\ y + \frac{\text{Reserve}(t+1)}{3} - \text{excess}(t) & \text{OTHERWISE} \end{cases}$$

This spear energy production changes during the day following the function:

$$\text{Reserve}(t) = \begin{cases} 3000 & \text{IF } t = 22(\text{mod}24) \\ 0 & \text{IF } t = 23(\text{mod}24) \\ 1000 & \text{IF } t < 4(\text{mod}24) \\ 1500 & \text{OTHERWISE} \end{cases}$$

- The technical features associated to the method used to generate the power.

$$\text{var}(t+1) = \begin{cases} \min(Z, \text{MaxI}, \text{MaxN}) & Z > 0 \\ \max(Z, \text{MaxD}, 0) & \text{OTHERWISE} \end{cases}$$

where MaxI is the maximum power increase per hour and MaxD is the maximum power decrease per hour.

Depending on the method used to produce the energy, these constants have the following values:

TYPE of POWER STATION	MaxN	MaxI	MaxD
Hydraulic	9000	5000	-5000
Nuclear	14000	3000	-2000
Thermic	18000	6000	-4000

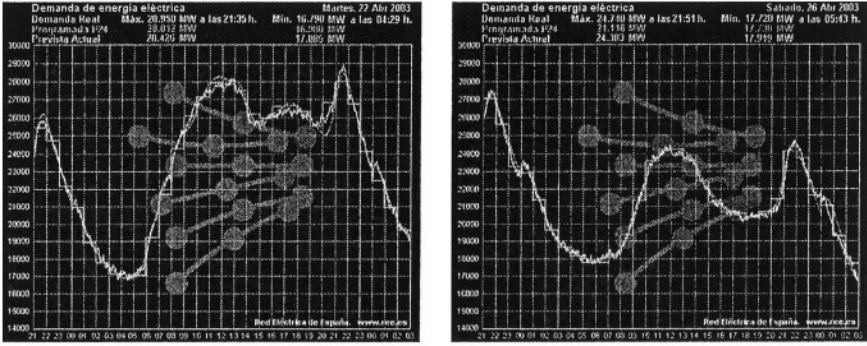


Figure 10.2. Example of demand on a labour day and a Saturday

So, the power produced by a power generation entity at time $t + 1$ will be:

$$\text{EnergyProd}(t+1) = \text{EnergyProd}(t) + \text{var}(t+1)$$

Notice that in this decision process we have not used at all the possible future demand of power.

4.2 Modelling the Demand

The demand has been modelled using as a reference the power consumption in Spain every hour during year 2001. The data has been taken from the “Red Electrica Española” which controls the electrical power distribution in Spain (see <http://www.ree.es>).

It can be observed that power consumption follows four different patterns: labour day, Saturday, Sunday and holidays. For example, Figure 10.2 shows the demand on a labour day and on a Saturday,

Using these four patterns we can simulate the demand of energy every hour. The demand for a week is computed using five working days + Saturday + Sunday and we substitute randomly (with a probability of 1/15) one of those days by a holiday.

Once decided which consumption pattern we will follow a specific day, to compute the demand of energy at time t we use the following formula:

$$\text{Demand}(t) = (\text{ConsPattern}(t) + \text{rand}(-250, 250)) \cdot (1 + \sin\left(\frac{\pi \cdot t}{4380}\right) 0.2)$$

where $\text{ConsPattern}(t)$ is the consumption at time t given the consumption pattern for that day (ConsPattern), $\text{rand}(-250,250)$ is a uniformly distributed random variable and $(1 + \sin\left(\frac{\pi \cdot t}{4380}\right) 0.2)$ allows us to model the variations in the consumption during the different seasons of the year with a variability over the basic pattern of $\pm 20\%$.

4.3 The Electrical Network System Operator

The system operator mediates between the producers and the consumers and also has authority to force producers to modify their production in order to satisfy the requirements of quality and security. In the EBM it distributes the demand among the producers using the following sequential procedure:

- If the produced power is less or equal than the demand then the system operator takes all the power production from each power station.
- If the production is bigger than the demand then:
 - 1 Distributes this demand among producers inverse proportion to the current performance of each power generation entity (where the performance is the ratio between the current production and the maximum capacity of a power generation entity).
 - 2 If there is still demand to be satisfied, then this demand is distributed in direct proportion to the spear power of each power generation entity.

This method minimizes for each power generation entity the difference between current and maximum capacity of production.

To plan the next step of production the power generation entities take into account the deficit or overproduction of the previous step as was explained in the previous section.

Notice that, in our model, producers decide about their own production and the system operator is responsible only of the demand distribution.

4.4 Power Cost

The prize of the power consumed is the total cost of the power produced including overproduction (that is, lost power that has not been consumed).

In the cost of production there is a maintenance cost (that do not depends on the quantity of power produced) plus the cost to produce each unit of power (GenerationCost).

We use Euros for cost and MWh for power production.

Our EBM uses the following costs:

TYPE	Maintenance cost (per hour)	Generation cost (per MWh)
Hydraulic	72000	16
Nuclear	224000	13
Thermic	180000	20

This costs have been obtained using the average cost of electrical power in Spain during year 2002 which is 38.91.

4.5 The EBM in Action

The properties that must fulfill the EBM are:

- That the deficit of power be punctual; and
- That the deficit of power never be greater than a 10% of the total production, which is the obliged reserve of power that each power generation entity must fulfill.

In the designed EBM, the average cost is 39.16 Euros/MWh that is very close to the real cost (38.91). The power lose (power that is produced but it is not consumed), is less than 8% of the total power consumed and the power deficits are punctual and never greater than 5%. Given these results we can say that the designed EBM fulfills our requirements.

5. Step 2: The Electronic Institution

The second step in the SADDE methodology consists on the design of an electronic institution that fixes a set of social interaction norms that are consistent with the relations established in Step 1.

The power electricity market we present in this section relies on a New Electricity Trading Arrangements (NETA) proposal presented the October 2000 in the United Kingdom¹.

Roles. There are three roles that an agent can play in the electricity market:

- **PRODUCERS:** Electricity producers that generate electricity using a different configuration of Power Stations. Power stations are expensive physical plants with a range of physical characteristics and running cost profiles.
- **CONSUMERS:** large industrial processes and local power distribution utilities.
- **SYSTEM OPERATOR (SO):** the operator of the energy transmission system, who is responsible for maintaining the supply voltage and system stability (preventing thermal overload and oscillation in flows - dynamic security).

The Markets. The electricity market is organized, in his turn, in several markets: the primary market, the secondary market and the balancing market. Finally there is a settlement stage.

¹The Spanish protocol is not yet available so we use the English model that is similar to what the Spanish market is expected to be.

- **PRIMARY MARKET:** There are periodic auctions (in our case every hour) of transmission rights, in the form of tickets valid for the injection or extraction of energy for an hour period. The auction protocol has not been specified, although a double auction seems likely. It is explicitly stated that the offer is greater than the demand.
- **SECONDARY MARKET:** Once the primary market for a specific period has been closed, the arrangements refer to the existence of an un-facilitated secondary market for the trading of transmission tickets. This market lasts until few minutes before the ticket becomes due. This time is known as “gate closure.” The objective for consumers and producers is to ensure they hold almost exactly the right number of tickets for each period of time to correspond to planned generation or expected consumption. For this market we propose a one to one negotiation mechanism as the procedure to exchange tickets.
- **BALANCING MARKET:** This market exists to permit the SO to maintain the voltage level and dynamic security. This market is performed once the secondary market closes. Based on its analysis of the transmission network and the flows as identified by tickets held, the SO can identify shortfalls or excesses of energy that will arise during the ticket window. The actions available to it are: (i) to dispatch additional generation; and (ii) to back-off scheduled generation.
- **SETTLEMENT:** in this stage consumers pay producers for the energy consumed.

These markets should run in parallel, that is, while in the primary market people is buying tickets for period T , in the secondary market they are negotiating tickets for period T' (where $T' > T$) and so on. However, to simplify, in our scenario we will consider a sequential order. Only one market is open at a time and they are opened sequentially. After the SETTLEMENT stage the cycle starts again with the PRIMARY MARKET.

5.1 ISLANDER Specification

An electronic institution model is based on four elements: dialogic framework, scenes, performative structure and norms. 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.

To illustrate how the electronic institution for the electricity market can be specified using ISLANDER we will show the scene that corresponds to the secondary market. In addition to that, the performative structure of the institution has a scene for the primary market, a scene for the balancing market and two more scenes (the root and the output scenes) that allow agents enter and leave the institution. Figure 10.3 shows a graphical representation using ISLANDER of the secondary market scene.

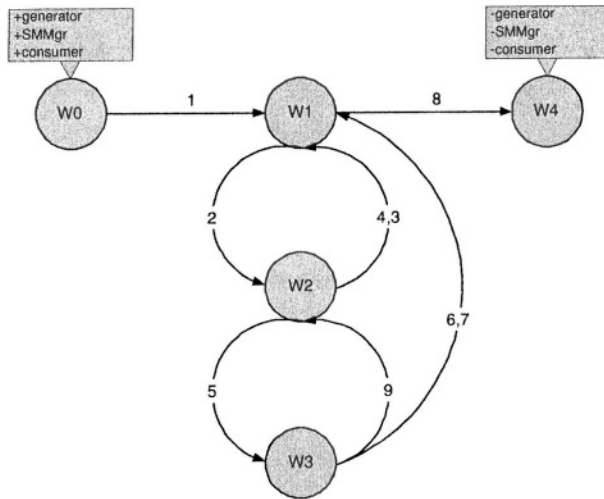


Figure 10.3. The secondary market scene

5.2 Specifying the Secondary Market Scene

Three different roles participate in the secondary market scene:

SMMgr: The secondary market manager. Only one agent playing this role can enter into the scene.

producer: A power producer.

consumer: A power consumer.

The secondary market scene contains five states, the *W0* state is the initial, and it is in this state where all the participating agents enter into the scene. When

all the agents are in the scene the *SMMgr* agent sends the (1) illocution to jump to the *W1* state where the negotiation between *consumers* and *producers* will be performed. When the *SMMgr* decides that the secondary market must be finished he sends the illocution (8) to inform that the market will be closed. After launching this illocution, the scene jumps to the *W4* state where all the agents leave the scene.

In *W1*, *W2* and *W3* states it is performed the negotiation between *consumers* and *producers*. When the scene arrives to the *W1*, each *producer* can start a negotiation process with a *consumer* sending an illocution (2) and jumping to the *W2* state. In this illocution the agent informs to the *consumer* about the quantity and price of the offer. At this state the *consumer* agent can do three actions:

- 1 Withdraw the offer: illocution (3);
- 2 Accept the offer: illocution (4); or
- 3 Generate a counter-offer: illocution (5).

If the *consumer* generates a counter-offer, the scene jumps to the *W3* state, where the *producer* can accept the last offer, cancel the negotiation, or make a new offer: illocutions 6, 7 and 9 respectively.

id	illocution	illocution constrains
1	(inform (?x SMMgr) (all all) start-secondary_market())	
2	(inform (?p producer) (?c consumer) offer(?quantity ?price))	(?quantity > 0 ?price > 0)
3	(inform (!c consumer) (!p producer) withdraw())	
4	(inform (!c consumer) (!p producer) accept(!quantity !price))	
5	(inform (!c consumer) (!p producer) offer(?quantity ?price))	(?quantity > 0 ?price > 0)
6	(inform (!p producer) (!c consumer) accept(!quantity !price))	
7	(inform (!p producer) (!c consumer) withdraw())	
8	(inform (!x SMMgr) (all all) end-secondary_market())	
9	(inform (!p producer) (!c consumer) offer(?quantity?price))	(?quantity > 0 ?price > 0)

The rest of the scenes that compound the performative structure are specified in a similar way.

6. Step 3: The ABM

The third step in the methodology consists on the specification of the decision making for the agents that will populate the electronic institution specified in the previous step. As we have said there are three types of agents that partic-

ipate in an electricity market: producers, consumers and the system operator. In the following sections we will describe each one of them.

6.1 Producers

Producers, as the name suggests, are responsible of generating the energy. In our example, energy can be generated using three different mechanisms: hydroelectric power stations, nuclear power stations and thermic power stations. Each type of power station is defined by four parameters:

- *PowerUp*: this parameter determines how much can be increased the production in this type of power station per time unit (in our case, one hour). This is a maximum and it is always conditioned to the total capacity of each concrete power station. For example, consider a power station that has a maximum production capacity of 800MW and the *PowerUp* for that kind of power stations is 200MW. If at time t that power station is producing 300MW it means that at time $t + 1$ it can generate no more than 500MW.
- *PowerDown*: the same meaning that the *PowerUp* parameter but associated to the decrease of production.
- *Cf*: cost per MW to maintain operative that kind of power station. To obtain the total cost for a concrete power station you have to multiply this value and the maximum production capacity. This cost is independent of the current production.
- *CMW*: how much it cost to generate a MW in that type of power station.

Each producer generates the energy using a different configuration of power stations. For example, you can have a producer that generates a 50% of the energy using nuclear power stations, a 30% using thermic power stations and a 20% using hydroelectric power stations. Taking into account this configuration and the parameters associated to each type of power station it is possible to calculate the capacity that has a given producer to increase or decrease the production per time unit (that is, the capacity to react to variations in the demand) and the cost of the produced energy.

In order to participate in the electricity market presented in section 5, a producer has to make decisions about three aspects:

- The amount of energy that will be generated during the next hour.
- The price and quantity of the energy offered in the double auction of the primary market.
- The negotiation process in the secondary market.

To decide the amount of energy to be generated during the next hour as well as the price to participate in the double auction of the next primary market, the producer uses a simple heuristic based on the result of its participation in the previous primary market. If in the previous round of the primary market the producer was able to sell the generated energy, then it will increase the production of each power station a quantity equal to *PowerUp* and will increase the price of the energy, otherwise it will decrease the production of each power station a quantity equal to *PowerDown* and will decrease the price of the energy. Concretely, producers use the following equations:

$$Q^t = \begin{cases} Q^{t-1} + \text{PowerUp} & \text{lastSold} = \text{true} \\ Q^{t-1} - \text{PowerDown} & \text{lastSold} = \text{false} \end{cases}$$

where Q^t is the quantity produced at time t and *lastSold* a flag that indicates if the producer was able to sell the produced energy the previous round of the primary market.

$$P^t = \begin{cases} P^{t-1} + \varepsilon^+ \cdot \frac{|P^{t-1} - AvP^{t-1}|}{AvP^{t-1}} & \text{lastSold} = \text{true} \\ P^{t-1} - \varepsilon^- \cdot \frac{|P^{t-1} - AvP^{t-1}|}{AvP^{t-1}} & \text{lastSold} = \text{false} \end{cases}$$

$$\text{Final}P^t = \min(P^t, \text{Production_cost})$$

where *Production-cost* is the cost to generate the energy (selling below that price means the producer is losing money), *Final* P^t is the price that will be uttered in the primary market at time t and AvP^{t-1} is the price that was paid in the primary market at time $t - 1$. *lastSold* has the same meaning that in the previous formula and ε^+ and ε^- are two constants particular to each producer.

Besides the energy it has decided to generate, the producer is obliged to generate always an extra amount of energy equal to the 10% of its maximum capacity. This energy cannot be sold in the primary and secondary markets and has to be available to the system operator in order to balance the market if it was necessary as we will show in section 6.3.

A producer always offers in the primary market all the energy that it has decided to generate (Q^t).

In order to simplify the analysis of the results, the negotiation process of the secondary market has been reduced to a double auction between two participants. The price uttered by the producer is a constant that is particular to each producer and the quantity is always equal to the amount of energy that was not sold in the primary market. This constant together with ε^+ , ε^- and the configuration of power stations are the set of parameters that define a producer's behaviour.

6.2 Consumers

The consumers that participate in an electricity market are companies that make a big consumption of energy, and energy wholesalers that later will sell the energy to smaller consumers.

Given the electronic institution presented in section 5, a consumer has to make decisions about three aspects:

- The demand of energy for the next hour.
- The offer to be uttered in the next double auction of the primary market.
- The negotiation process in the secondary market.

The demand of energy is modelled using real data. This data and the algorithm to generate the demand are the same that are used in the EBM model. Once determined the demand for the next hour, this demand is distributed equitably among all the consumers. Then, each consumer individually will try to cover that demand.

The strategy to participate in the double auction in the primary market is very similar to the one used by producers. The price is determined by the formula:

$$FinalP^t = \begin{cases} P^{t-1} - \epsilon^+ \cdot \frac{|P^{t-1} - AvP^{t-1}|}{AvP^{t-1}} & lastBought = true \\ P^{t-1} + \epsilon^- \cdot \frac{|P^{t-1} - AvP^{t-1}|}{AvP^{t-1}} & lastBought = false \end{cases}$$

where $FinalP^t$ is the price that will be uttered in the primary market at time t , AvP^{t-1} is the price that was paid in the primary market at time $t-1$, $lastBought$ is a flag that indicates if the consumer was able to buy in the previous round of the primary market and ϵ^+ and ϵ^- are two constants particular to each consumer. The quantity of energy requested is equal to the demand for the next hour.

As we have explained for producers, the negotiation process of the secondary market has been reduced to a double auction between a single producer and a single consumer. The price uttered by the consumer is again a constant that is particular to each consumer, and the quantity is always equal to the amount of energy that the consumer still requires to fulfill the demand after the primary market.

The price of the offer in the secondary market and the constants ϵ^+ and ϵ^- are the parameters that define a consumer's behaviour.

6.3 System Operator

The task of the SO is to maintain the voltage level and dynamic security of the electricity network. Based on its analysis of the results in the primary

and secondary markets, the SO can identify future shortfalls or excesses of energy and try to avoid them. After analyzing the situation after the primary and secondary markets there are several possibilities:

- The demand has been covered in the primary and secondary markets. The system operator notifies to the producers with energy that was not sold in the primary and secondary markets that they have to disconnect their power stations from the network. This measure is to avoid an overload that could be dangerous for the integrity of the network.
- There is some demand that was not covered after the primary and secondary markets. First, the system operator tries to cover the demand with the energy that will be generated but that was not sold in the primary and secondary markets. If after that there is still demand to be covered, the system operator uses the 10% of extra energy that each producer is obliged to generate specially for these occasions.

The price that will be paid for the energy assigned by the system operator is the average cost of all the energy that is going to be generated by producers (including the energy that has not been sold).

7. Step 4: Multiagent System

The final step is to build the MAS that implements the electronic institution presented in section 5 and it is populated by instances of the agents presented in section 6.

In our experiments we do not have real agents running in parallel and exchanging messages through a communication platform. Instead, agents are implemented as C++ classes and the exchange of messages is done through method calling. This allows us to run the experiments very fast without compromising the validity of the obtained results.

8. Cycle P4 through Evolutionary Computing

To explore the different configurations of agent instances to build the MAS and to find which of those configurations satisfy the requirements fixed by the EBM in the first step of the methodology, we propose the use of evolutionary computing. In the following sections we will explain in detail how this approach is applied in the electricity market scenario.

8.1 Gens and Chromosomes

A MAS is fully specified by a chromosome. Each gene of that chromosome contains the information that defines an agent. In the electricity market

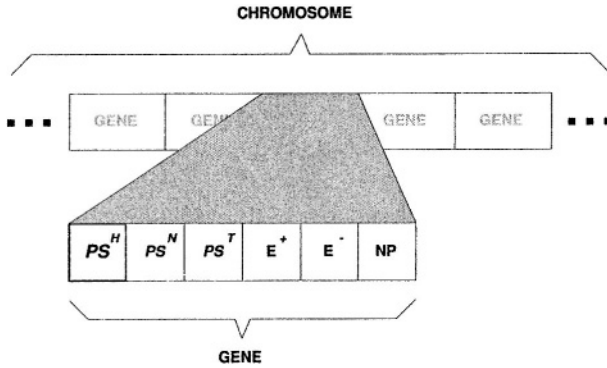


Figure 10.4. A chromosome for the electricity market

scenario there are two types of agents that we want to explore: producers and consumers.

As we have said there are five parameters that specify a producer:

- The configuration of power stations that determines how the producer generates the energy. There are three parameters, each one representing the different types of power stations: Hydroelectric noted as PS^H , Nuclear noted as PS^N and Thermic noted as PS^T .
- ϵ^+ and ϵ^- that define the strategy of the producer to fix the prices in the double auction (primary market).
- The negotiation price, noted as NP , that defines the price the producer will offer in the secondary market.

and three parameters that specify a consumer:

- ϵ^+ and ϵ^- that define the strategy of the consumer to fix the prices in the double auction (primary market).
- The negotiation price, noted as NP , that defines the price the consumer will offer in the secondary market.

To simplify the genetic operations (mutation, crossover, etc.) we have unified the length of the genes in the chromosome. Parameters PS^H , PS^N and PS^T are used also to represent a consumer but only for syntactical reasons and they are always equal to 0. Figure 10.4 shows a chromosome codifying a MAS for the electricity market.

It is important that the genes of the chromosome fulfill the following restriction:

$$\sum_{i \in n} PS_i^H = 1$$

$$\sum_{i \in n} PS_i^N = 1$$

$$\sum_{i \in n} PS_i^T = 1$$

where n is the number of genes. To achieve that, we have to normalize these parameters in the chromosome after applying every genetic operator.

8.2 Fitness Functions

We use the aggregation of three different functions to evaluate the fitness of an individual (MAS):

1

$$f1 = ac \left(\frac{100 \cdot |AvABM - AvEBM|}{AvEBM} \right)$$

This function computes how near is the average cost of the electricity produced in the MAS ($AvABM$) with the average cost of the electricity we have calculated using the EBM ($AvEBM$). The ac function is defined as:

$$ac(x) = \begin{cases} 1 & \text{IF } x < 8 \\ 1 + \frac{(8-x)}{7} & \text{IF } 8 \leq x \leq 15 \\ 0 & \text{OTHERWISE} \end{cases}$$

2

$$f2 = ed \left(\frac{100 \cdot PowerDeficit}{Production} \right)$$

This function computes the percentage of power deficit relative to the total production. The ed function is defined as:

$$ed(x) = \begin{cases} 1 & \text{IF } x < 1 \\ \frac{(3-x)}{2} & \text{IF } 1 \leq x \leq 3 \\ 0 & \text{OTHERWISE} \end{cases}$$

3

$$f3 = pl \left(\frac{100 \cdot PowerLost}{Production} \right)$$

This function evaluates the power lost in our model (power that is produced but is not consumed). The pl function is defined as:

$$pl(x) = \begin{cases} 1 & \text{IF } x < 8 \\ \frac{(10-x)}{2} & \text{IF } 8 \leq x \leq 10 \\ 0 & \text{OTHERWISE} \end{cases}$$

The formula to compute the fitness function is:

$$Fitness = 0.4 \cdot f1 + 0.3 \cdot f2 + 0.3 \cdot f3$$

```

numInd      = 50      // Number of individuals in each population
xoverpr     = 0.1     // Crossover probability
mutpr       = 0.05    // Mutation probability
fitnessThr  = 0.9     // Threshold for the termination condition
minimumFit  = 0.7     // Minimum acceptable fitness for the
                    // termination condition
EBMAVCost   = 39.16   // Average price of the electricity in the EBM
ProdDeficit = 5       // % of demand not fulfilled
                    // (relative to the total production)
LostPower   = 8       // % of power that is produced but it is not
                    // consumed (relative to the total production)

```

Figure 10.5. Parameters for the genetic algorithm

8.3 Description of the Experiments

The aim of the experiments is to find a MAS that converges with the EBM in three aspects:

- The average price of the electricity in the market during the analyzed period. We have analyzed the market using the EBM during 720 iterations (equivalent to one month) and the average price obtained was 39.16. This is the value that we want to achieve using the ABM.
- The percentage of demand, relative to produced energy, that cannot be fulfilled. Here we will not tolerate a percentage greater than 5%.
- The percentage of power relative to the total production that is lost (i.e., power that is generated but it is not consumed). In the EBM we found that the amount of power produced that is not consumed was about an 8%.

The general parameters of this experiments are shown in Figure 10.5. The termination condition imposed to the genetic algorithm is that the best individual of the population have to fulfill the following conditions:

- The average fitness after 15 executions of this individual must be greater than *fitnessThr*.
- The minimum fitness after this 15 executions must be greater than *minimumFit*.

This restrictions guarantee that the individuals are good and robust enough.

EupProd and *EdownProd* are the possible range for parameters ϵ^+ and ϵ^- respectively in the case of producers and *EupCon* and *EdownCon* are the possible range for parameters ϵ^+ and ϵ^- respectively in the case of consumers (see section 6.1 and section 6.2).


```

numRounds = 720 // number of electricity market sessions

PRODUCERS:
numProd      = 30 // number of producers
EupProd      = [0.05,0.15]
EdownProd    = [0.05,0.15]
NegoProd     = [25,60]

CONSUMERS:
numCon       = 60 // number of consumers
EupCon       = [0.05,0.15]
EdownCon     = [0.05,0.15]
NegoCon      = [10,45]

```

Figure 10.6. Parameters for the MAS

NegoProd is the possible range for the negotiation price in the secondary market for producers and *NegoCon* is the possible range for the negotiation price in the secondary market for consumers.

Each electricity market session implies the execution of the primary market, the secondary market and the balancing market plus the settlement of the exchanged tickets (generation and payment of the energy). Because this happens every hour, by running 720 sessions we are simulating one month ($720 / 24 = 30$).

The fitness function used in these experiments is the one presented in section 8.2, that is:

$$\text{Fitness} = 0.4 \cdot f1 + 0.3 \cdot f2 + 0.3 \cdot f3$$

where $f1$ computes how near is the average cost of the electricity produced in the MAS, $f2$ computes the percentage of power deficit relative to the total production and $f3$ evaluates the power that is produced but is not consumed.

8.4 Results

In all the experiments, after about 20 generations the genetic algorithm was able to find an individual that fulfilled the requirements established by the EBM and described in previous sections.

9. Conclusions

EBM and ABM are two well known styles of computer based modeling. EBM has a long tradition and a selection of friendly tools, ABM is a more recent but a powerful approach. EBM allows the modeling of the global behaviour of a population leaving implicit the behaviour and interaction of in-

dividuals. On the other hand in ABM we model explicitly these individuals and their interactions leaving the global behavior of the population as an emergent result. There are numerous applications of each of these approaches (Rodríguez et al., 1998; Ruth and Hannon, 1997). They have even been applied to the same problem in order to establish comparative criteria about their alternative use (Parunak et al., 1998a). This competing view between EBM and ABM makes sense if you have a real system against which the model you build should be checked. However if the goal is to build an artificial system whose behavior is to be inspired by a real system but not bound to simulate it faithfully, then the reasonable attitude is to take EBM and ABM as complementary approaches to be used at different levels of abstraction in the design lifecycle.

We have integrated both approaches into a methodology for MAS design and implementation. More specifically we have used EBM to identify desired global properties of the MAS. Then we analyzed how the flows of the EBM could be produced by the interactions between different types of agent. The structure of the EBM guides the definition of these interactions through an electronic institution. We then decide on the agent model we expect that will allow populations whose aggregate behavior will meet the EBM. Finally, the model parameters become the genes of agents in the MAS when exploring the space of models using evolutionary computing.

The application of evolutionary programming to MAS redesign brings us several preliminary results.

- The chosen agent model allows the convergence of the evolutionary process towards the production of a stable collection of MASs showing the EBM specified properties to an acceptable degree. This is so in a huge search space. This shows that the choice of GAs as a basic technique for SADDE is justified.
- We found that once a good ABM model is engineered to match the behaviour predicted by the EBM, it is not easy to populate a MAS by fixing the agent parameters. Initial populations with randomly fixed parameters show a very bad global behaviour. This can be due to the fact the the ABM model does not impose enough constraints on the behaviour of agents and therefore the space contains big areas where the behaviour is not good enough.
- According to several robustness experiment we see that once a stable population is found up to a 20% random changes in the individuals can be tolerated within acceptable ranges in the fitness function.
- The fact that a GA converges starting from an initial MAS at Step 4 of the SADDE methodology proves that the model chosen for the ABM at

step 3 in SADDE is feasible to show the behaviour specified at the EBM. A failure on this would mean that the ABM model is not appropriate.

- Finally, we have observed through experimentation that societies with a high interaction degree show more stable behaviours.

We intend to continue the research in the area of complex systems as we are convinced that if a robust methodology has to be found to develop MAS it has to deal with the highly non-linear problems that systems composed of thousands of autonomous entities raise. Agents have lost the composability that was natural in all computing paradigms from functional to imperative to object oriented. Autonomy has open the door to many unanswered questions that currently prevent from having robust methodologies.

Chapter 11

THE PROMETHEUS METHODOLOGY*

Michael Winikoff and Lin Padgham

Abstract In this chapter we present the Prometheus methodology for building agent-based software systems. Our goal in developing Prometheus was to have a process with associated deliverables which could be used by industry practitioners and undergraduate students without a previous background in agents. As a result, the Prometheus methodology aims to be detailed and complete, as well as being general-purpose and having tool support. Prometheus comprises three phases: system specification, architectural design, and detailed design. The Prometheus methodology has been developed over a number of years as a response to both educational and industrial needs. The methodology has been used by industrial practitioners, taught at workshops at a number of conferences, and has been taught to undergraduate and postgraduate students, as well as having been used in student projects. These experiences have been positive and we have noticed an enormous difference in the ability of our students to develop agent systems. Using Prometheus third year undergraduates are able to build reasonable agent systems in a one semester course, something that previously was challenging for graduate students.

Keywords: Agents, Software Engineering, Methodologies.

1. Introduction

*Prometheus*¹ is a methodology for developing agent-oriented software systems. Our goal in developing Prometheus was to have a process with associated deliverables which could be used by industry practitioners and undergraduate students to develop intelligent agents systems, without a previous background

* Figures 11.1 and 11.2, the Query Late Books Scenario, and some of the text in the Architectural Design section are reproduced by permission of John Wiley & Sons, Ltd. from Lin Padgham and Michael Winikoff, *Developing Intelligent Agent Systems: A Practical Guide to Design*, ISBN 0-470-86120-7 to be published in 2004.

¹Prometheus was the wisest Titan. His name means “forethought” and he was able to foretell the future. Prometheus is known as the protector and benefactor of man. He gave mankind a number of gifts including fire (from <http://www.greekmythology.com>).

in agents. To this end Prometheus aims to be *detailed* and *complete* in the sense of covering all the stages of software development as applied to agent systems.

The Prometheus methodology includes three phases:

- The *system specification phase* focuses on (i) identifying the system's *interface*, that, since we are dealing with situated agents, consists of *percepts* (information from the environment), and *actions*; and (ii) determining the system's goals, functionalities, and use case scenarios, along with any important shared data. The outputs from this phase are a set of functionality descriptions, percept and action descriptions, system goals, and use case scenarios.
- The *architectural design phase* uses the outputs from the previous phase to determine which agents the system will contain, how they will interact, and what significant events occur in the environment. The outputs of this phase are a system overview diagram, agent descriptions, agent interaction protocols and a list of significant events and messages between agents.
- The *detailed design phase* looks at the internals of each agent and how it will accomplish its tasks within the overall system. The outcomes of this phase are detailed diagrams showing the internal functionality of each agent and its capabilities, process diagrams that show the internal processing of the agent, as well as descriptions of data structures used by the agent, plans and subtasks and the details of plan triggers.

Figure 11.1 indicates the main design artifacts that arise from each of these phases as well as some of the intermediary items and relationships between items. The figure shows the models and dependencies, but does not show the process (although it does imply it).

The development (and revision) of the various models depicted in Figure 11.1 is intended to proceed in an iterative fashion (similar to the Rational Unified Process) where in each iteration the focus of the work gradually shifts further down towards implementation, but where it is expected that most iterations will not be exclusively concerned with a single phase and that many iterations will involve revision of previously developed models.

Figure 11.1 is divided horizontally and vertically. The three horizontal regions form the three *phases* of the methodology discussed above. The left-most region (consisting of scenarios, interaction diagrams, interaction protocols and process diagrams) deals with descriptions of the *dynamic* behaviour of the system. The middle vertical region (data coupling, acquaintance, system overview, agent overview and capability overview) deal with *overviews* of the system while the remaining models (the right region) give detailed descriptions

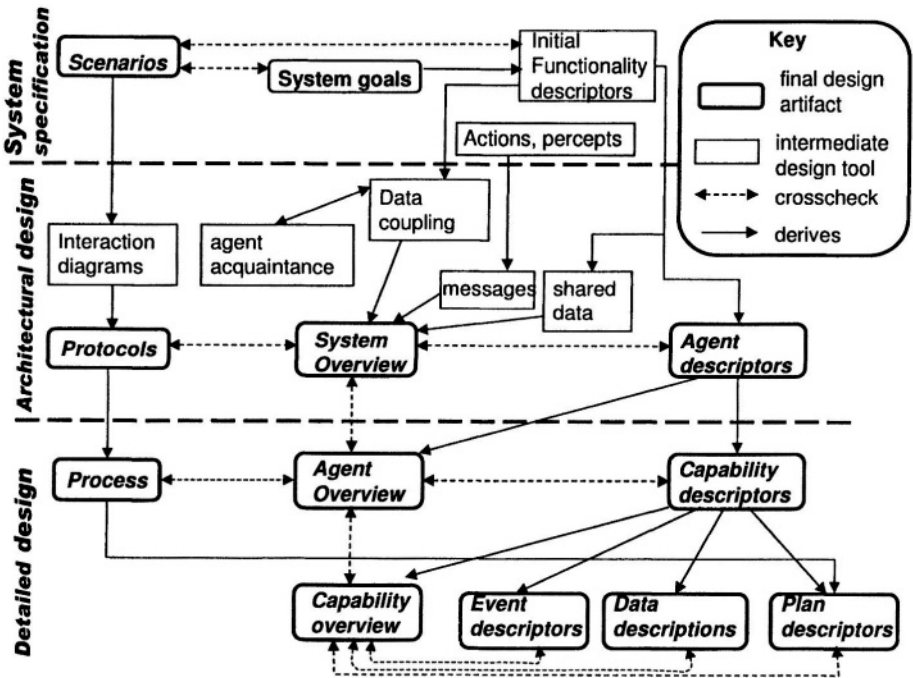


Figure 11.1. Overview of the Prometheus Methodology

for each entity in the system. Both the middle and right region deal with the static structure of the system.

Prometheus, like any other methodology, defines a number of system models and notations that are used to describe these models. We describe structural overviews at various levels (system, agent, capability) with a single diagram type. In addition, diagrams are used for showing data coupling and agent acquaintance relationships. Dynamic behaviour is currently described with existing models from UML (Unified Modeling Language) and AUML (Agent UML) (see chapter 12).

In addition to graphical notations, we use structured textual descriptors (i.e., forms) for describing individual system entities (e.g., agents, functionalities, plans, etc.). We also maintain a data dictionary which is important in ensuring consistent use of names.

It is important to note that Prometheus is a general purpose methodology. In particular, most of the methodology (specification and architectural design) does *not* assume a particular agent architecture. Although the detailed design phase does target a particular family of agent architectures (namely those that achieve goals using a library of plans), this does not make Prometheus special-

purpose. Any methodology that addresses implementation needs to have a target platform. For example, Tropos (Bresciani et al., 2002) (see chapter 5) also targets BDI-like systems, whereas Gaia (see chapter 4) avoids the issue by not addressing implementation.

In the following sections we briefly describe the processes and models associated with each of the three phases. Due to space limitations and the desire to describe all of the methodology this chapter cannot do justice to Prometheus. In particular, we cannot describe a running example in detail, and the detailed techniques, that is *how* particular steps in the process are performed, are not described. The description in this chapter is current as of October 2003. For further or up-to-date information see <http://www.cs.rmit.edu.au/agents/SAC>.

2. System Specification

System specification consists of three main activities: determining the system's interface to the environment, determining the system's goals and functionalities, and determining scenarios which capture the usage of the system.

Since agents are situated, one of the key things to be captured in the development process is how the agents interact with their environment. Following standard terminology (Russell and Norvig, 1995) we call incoming information from the environment *percepts* and agents' means of affecting the environment *actions*. As discussed in (Winikoff et al., 2001) the raw data from percepts may need to be processed in order to obtain things that are a significant event for the agent system. Prometheus prompts the developer to consider such issues. For example a video frame from a camera on a soccer playing robot, may need processing to extract the symbolic objects, such as ball, goal and players, as well as further processing to determine whether anything significant has actually happened – such as a ball having moved since a previous frame, or a ball not appearing where one was expected.

Determining the system's goals and functionalities is done by iterating over the following steps:

- Identify and refine system goals – main and subsidiary;
- Group goals into functionalities;
- Prepare functionality descriptors;
- Define use case scenarios (and variations); and
- Check that all goals are covered by scenarios.

An initial set of goals is identified from the initial requirements. These are refined and elaborated into a hierarchy of goals by asking *how* goals will be

achieved, and *why*² goals are being achieved (van Lamsweerde, 2001). For example, if we are designing an online book store we might have a high-level goal *fully online system*. This goal might have associated with it the subgoals *find books online*, *pay online* and *order online*.

Functionalities are limited “chunks” of system behaviour that describe in a broad sense what the system needs to be able to do. We derive functionalities by grouping related goals. For example, given the goals above, we might also have another high-level goal of *purchase books* with subgoals *find books*, *place order*, *make payment*, and *arrange delivery*. *Pay online* and *make payment* are clearly closely related if not identical goals, and are therefore grouped together in a single functionality.

Functionality descriptors capture the name and description of each functionality as well as what events activate it, what goals it achieves, what actions it performs, what percepts it receives, what messages it sends/receives, and what data it uses and produces.

Use case scenarios are complementary to goals in that they show how processes are composed within the system. In developing goals, we typically already are building up scenarios of how these goals will be part of various processes within the system. Scenarios enable us to specify some of this structure, which in turn may help to identify missing goals.

Use case scenarios are based on ideas from object oriented design but are more structured. This structure allows for automated cross checking, and automatic production of partial information for later design artifacts (e.g., protocols).

The core of the use case scenario is the sequence of steps describing a particular example of the system in operation. Each step can optionally have data read and data written noted as well as the functionality that performs that step. Each step can be a GOAL, ACTION, PERCEPT or SCENARIO, as well as OTHER allowing for additional step types, although these cannot be used in automated processing. The following example illustrates the steps of a use case scenario in Prometheus.

Query Late Books Scenario

Trigger: User enquiry

1. GOAL: Determine delivery status
2. GOAL: Log delivery problem
3. ACTION: Request delivery tracking
4. GOAL: Inform customer
5. OTHER: Delay

²At the moment we do not use this in Prometheus – asking “why” can help derive early requirements that capture why the system is being built.

6. PERCEPT: Tracking information received
7. GOAL: Arrange delivery
8. GOAL: Log books outgoing
9. GOAL: Inform customer
10. GOAL: Update delivery problem

Functionality descriptors, goals, and use cases give different views of a common underlying system. As a result they should be checked for mutual consistency. For example an interaction between functionalities in a use case scenario should also be evident in the interactions field of a functionality descriptor. Also, each system goal should be represented in at least one scenario; all functionalities should be covered; and use case scenarios should cover the important normal uses of the system as well as some error/unusual situations, in order to give an idea of how these will be handled.

3. Architectural Design

The three aspects that are developed during architectural design are:

- 1 Deciding on the *agent types* used in the application. Agent types are formed by grouping a number of functionalities together. Diagrams which we use to assist in the analysis are *data coupling diagrams* and *agent acquaintance diagrams*.
- 2 Designing the overall system structure (with a *system overview diagram* along with descriptors).
- 3 Describing the interactions between agents using *interaction diagrams* (developed from scenarios) and *interaction protocols* (developed from interaction diagrams).

3.1 Deciding on the Agent Types

One technique that we use to systematically examine the properties which lead to coupling and cohesion is the *Data Coupling Diagram*. Potential groupings are then evaluated and possibly refined using an *Agent Acquaintance Diagram*.

A data coupling diagram (see Figure 11.2) consists of the functionalities and all identified data (not only persistent data, but also data the functionalities require to fulfil their job). Directed links are then inserted between functionalities and data, where an arrow pointing towards the data indicates the data is *produced or written by* that functionality, whereas an arrow pointing towards the functionality indicates the data is *used by* the functionality. A double-headed arrow indicates that the functionality both uses and produces the data. Edges

between data and data or between functionality and functionality are incorrect syntax (and cannot be drawn in the tool).

The data coupling diagram is used to identify groupings which are linked by their data use. When assessing the diagram visually we are looking for clusters of functionalities around data. This is one important aspect in the analysis of potential groupings of functionalities. It is also used to guide refinements and changes to achieve a cleaner delineation between agents.

Some reasons for grouping functionalities into a single agent are if the functionalities seem to be related or if they share a lot of information. Some reasons for *not* grouping functionalities are if the functionalities are clearly unrelated, or if they exist on different hardware platforms.

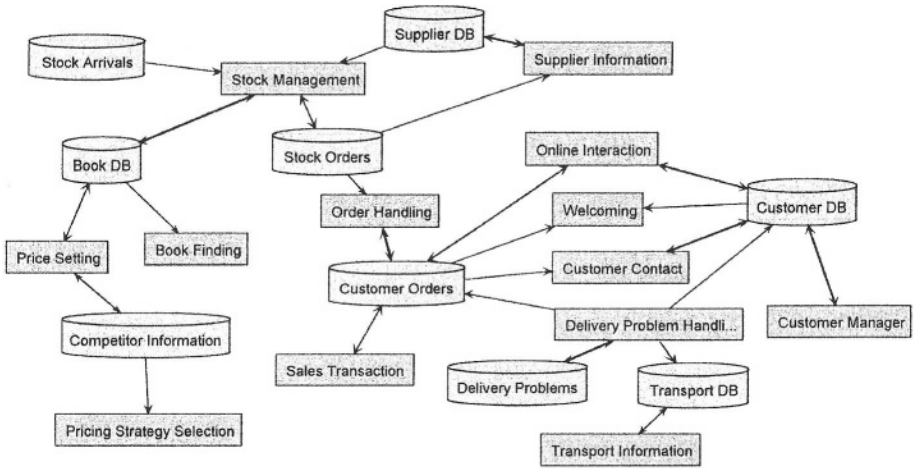


Figure 11.2. Data Coupling Diagram

In order to evaluate a potential grouping of functionalities into agents with respect to agent coupling we use an *agent acquaintance diagram* (see Figure 11.3). This diagram represents each of the agent types in the system. Information about agent interaction is extracted from the functionality descriptors and each agent type is linked with the other agent types it interacts with. Links can be decorated with the cardinality of the relationship if desired (e.g., one warehouse agent interacts with many sales agents).

We then analyse the resulting diagram in two ways. One is simply an analysis of the density of the links within the diagram. It is a measure of the ratio of the actual coupling to the maximal possible coupling. If the system has four agents, then each agent could potentially be linked to a maximum of three other agents, giving a total number of 3 + 2 + 1 possible links. To get the link

density we simply count the links and divide by this number. This measure is only one aspect of the analysis. We also consider bottlenecks and other issues.

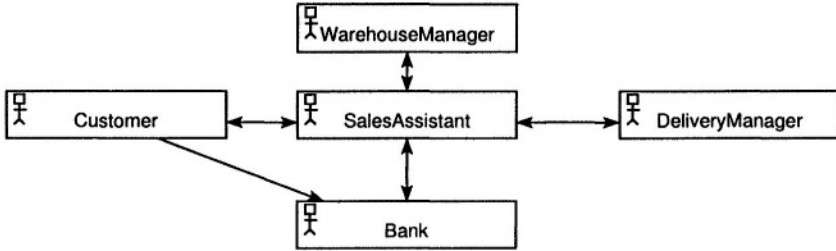


Figure 11.3. Agent Acquaintance Diagram

3.2 Designing the Overall System Structure

The system overview diagram is arguably the single most important artifact of the entire design process, although of course it cannot really be understood fully in isolation. The various descriptors provide the more detailed information that may be required.

The notation used in the system overview diagram (and in agent and capability overview diagrams) is a directed graph where nodes represent design entities and directed arcs represent relationships. Figure 11.4 depicts the nodes that are currently used, these correspond directly to the concepts used in the Prometheus methodology.

A syntactically valid overview diagram consists of a set of nodes (excluding goals and functionalities), each labelled with a name, with links between them. We distinguish between “active” nodes (entities that do things – agents, capabilities, and plans) and “passive” nodes (anything else – percepts, actions, messages, protocols, data). A link is valid from an active node to a passive node or from a passive node to an active node. A link is *not* valid from an active to an active node or from a passive to a passive node. An additional

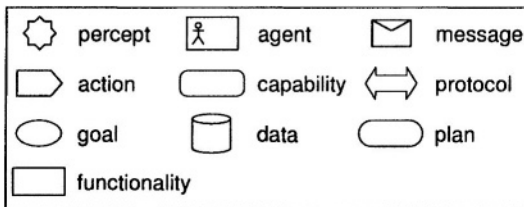


Figure 11.4. Notation used in Overview Diagrams

constraints is that there cannot be links *to* a percept and there cannot be links *from* an action.

The meaning of links is as follows:

- A link *to* a message indicates that the agent type (or capability or plan) sends that message.
- A link *to* a protocol indicates that the entity communicates using the protocol in question.
- A link *to* an action indicates that the entity performs the action.
- A link *to* a data node indicates that the entity writes to it.
- A link *from* a message indicates that the agent type (or capability or plan) receives that message.
- A link *from* a percept indicates that the entity receives the percept.
- A link *from* a data node indicates that the entity reads the data.

When drawing the system overview diagram (see Figure 11.5) we start by creating a named agent symbol for each agent type. We also add the percepts and actions at this point.

A data store icon is placed for each persistent data store, with an incoming link from each agent that writes to the data store and an outgoing link from the data store to each agent that directly accesses the data. Double headed links (arrows at both ends) indicate both read and write.

Once interaction protocols have been defined they are added into the diagram and we indicate which agents participate in these protocols.

3.3 Describing the Interactions between Agents

This sub-phase focuses on the system's *dynamic* behaviour by fully specifying the interaction between agents. *Interaction diagrams* borrowed from UML sequence diagrams, are used as an initial representation of agent interaction. Fully specified *interaction protocols* (borrowed from the revised version of AUML currently under development) are the final design artifact.

Interaction diagrams are the same as sequence diagrams of UML except that they show interaction between agents rather than objects. One of the main processes for developing interaction diagrams is to take the use case scenarios developed in the specification phase and to build corresponding interaction diagrams, showing the interaction between agents in a scenario.

As with scenarios, we would expect only to have a representative set of interaction diagrams, not a complete set. In order to have complete and precisely

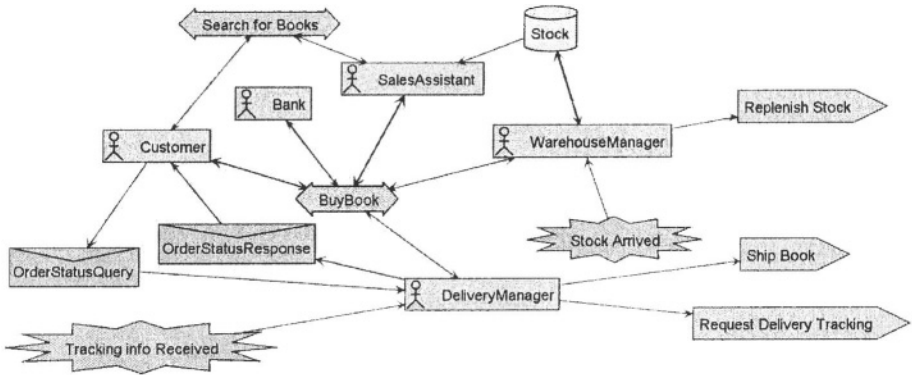


Figure 11.5. System Overview Diagram (excerpt)

defined interactions we progress from interaction diagrams to protocols which define exactly which interaction sequences are valid within the system.

Developing protocols is done by considering alternatives. For each message (or percept) that an agent receives we ask “*what are the possible messages that the agent could send as a response?*” We then repeat the process for these messages. Because protocols must show all variations they are often larger than the corresponding interaction diagram and may need to be split into smaller chunks.

An example interaction diagram can be found in Figure 11.6 and an example interaction protocol (using the new AUML notation) can be found in Figure 11.7.

4. Detailed Design

This phase deals with the internals of each agent, rather than the system as a whole. We use a hierarchical model so that each agent is broken up into capabilities. Capabilities may be included in more than one agent.

The steps within detailed design are:

- 1 Develop agent overviews (showing interactions between capabilities) and capability descriptors.
- 2 Develop the internal process of an agent from the interaction protocols, described using a variant of UML activity diagrams³.

³This is not discussed further in this chapter.

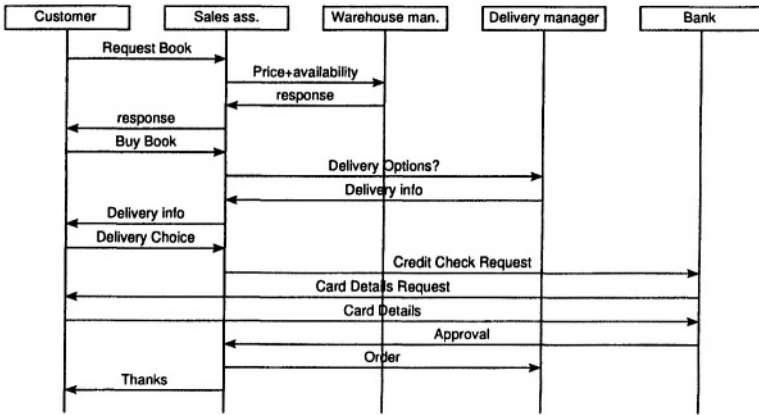


Figure 11.6. Interaction Diagram

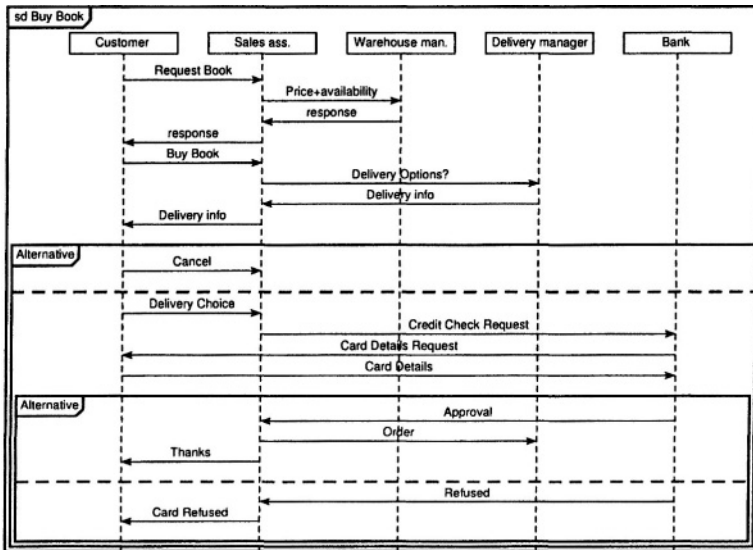


Figure 11.7. Interaction Protocol

- 3 Develop the internal design of each capability in terms of plans, events, beliefs, and (possibly) sub-capabilities.

The process followed is essentially iterative refinement. We begin by considering for each agent what the agent needs to be able to do. Often, the functionalities that were grouped to form the agent type will be a good starting point for defining the capabilities of that agent type.

We then “connect” up the capabilities. As depicted in the system overview diagram, each agent has incoming and outgoing messages, percepts that it received, actions that it performs, and data that is read and/or written. Each of these connections to an agent is mirrored in the *agent overview diagram* for that agent type. The agent overview diagram for a given agent type is quite similar to the system overview diagram, but shows interactions between capabilities *within* an agent, rather than between agents within a system. Any messages or percepts that are incoming to an agent in the system overview, must be incoming to some capability (or plan) within that agent in the capability overview diagram. Similarly any actions or messages that are outgoing from an agent in the system overview, must be outgoing from some capability (or plan) within that agent in the capability overview diagram.

Once capabilities (and plans) within an agent have been defined we consider each capability and refine its internals. This process continues until the internal operation of each agent and each capability is defined in terms of plans, messages, data, and other capabilities.

5. Tool Support

Designs for large systems are almost always developed incrementally with many revisions. When revising any artifact, be it documentation, code, or design, it is easy to introduce inconsistencies and minor errors. We have found tool support to be extremely useful for checking and maintaining design consistency across varying levels of detail.

The *Prometheus Design Tool* (PDT) allows a user to enter and edit a design, in terms of Prometheus concepts; check the design for a range of possible inconsistencies; and automatically generate a design report that includes descriptors for each design entity, a design dictionary, and the various diagrams. It also provides descriptor forms which prompt for the various aspects which should be considered. When any aspect of the design is modified, the change is propagated to all levels, although in some cases user input is still required for finalisation.

PDT supports the Prometheus methodology in a number of ways. It supports the process of deriving agent types from functionalities by deriving part of each agent’s interface, by cross checking the declared interface of an agent against the functionalities that make up the agent type, and by generating cou-

pling and acquaintance diagrams. It supports the process of developing the internals of agents in the detailed design phase by cross checking an agent's internals against the agent's declared interface, checking the consistency of a plan with its context, and by supporting views of design diagrams at different levels (system overview, agent overview, and capability overview). For more details on tool support for the Prometheus methodology see (Padgham and Winikoff, 2002) (this paper discusses an early prototype tool which was also, somewhat confusingly, called PDT).

The Prometheus Design Tool is currently available⁴ and further functionality is under development.

5.1 Debugging with Design Artifacts

The Prometheus methodology aims to support the full life cycle, including testing and debugging. David Poutakidis, a research student of the authors, has been working on debugging MAS, using design artifacts such as those produced by the Prometheus methodology. The central claim in his work is that:

"... design documents and systems model developed when following an agent-oriented software engineering methodology [such as Prometheus] can be incorporated in an agent and used at run-time to provide for run-time error detection and debugging." (Poutakidis et al., 2002)

Specifically, the work described in (Poutakidis et al., 2002; Poutakidis et al., 2003) uses *interaction protocols* expressed in AUML (Odell et al., 2000). These are translated into Petri nets and a debugging agent uses these to monitor agent interactions and alert the programmer when a protocol is not followed correctly.

5.2 Code Generation

The latest version of the JACK⁵ Development Environment (JDE) includes a design tool that allows Prometheus overview diagrams (based on a slightly older version of the methodology) to be drawn. The JDE also includes a graphical user interface that allows the structure of an agent system to be built by drag-and-drop and by filling in forms.

The JDE supports the Prometheus methodology in that the concepts provided by JACK correspond to the artifacts developed in Prometheus' detailed design phase. It is important to realise that the agent structure described in the JDE generates JACK code that can be compiled and run. This automatic

⁴PDT can be downloaded from <http://www.cs.rmit.edu.au/agents/pdt>

⁵JACK is a commercial agent development platform developed by Agent Oriented Software. It includes an agent-oriented programming language that is a superset of Java.

generation of skeleton code from design artifacts is extremely useful, and has encouraged students to do design prior to coding.

6. Experiences with Using Prometheus

The Prometheus methodology has been developed over a number of years, as a response to both educational and industrial needs. During its development it has been used by industrial practitioners, taught at workshops at a number of conferences, and has been taught to undergraduate and postgraduate students, as well as having been used in student projects.

We have worked with development of agent software for eight years and have during this time had a wealth of experience in trying to teach students to build such systems. The *Prometheus* methodology has partially grown out of this experience and we have noticed an enormous difference in the last few years, in the ability of our students to develop agent systems. Previously, without a methodology, graduate students would flounder and end up building a system which made little real use of agents. Using Prometheus, third year undergraduates are now able to build reasonable agent systems in a one semester course.

We have worked with companies that sell agent development platforms (for BDI agents) and they have experienced similar difficulties with teaching their customers how to develop agent based systems, as we have experienced with students. We have worked with Agent Oriented Software⁶ (AOS) both in developing the methodology and also in producing materials for training professional software developers in development of agent systems. The methodology has formed the basis for a course on agent-oriented design that is offered by AOS to industry software developers who are starting to use the JACK intelligent agents development environment (Busetta et al., 1998), and has been successful in introducing them to methods to assist them in design of agent applications. For example, a prototype weather alerting system (Mathieson et al., 2004) developed for the Australian Bureau of Meteorology by AOS used Prometheus overview diagrams (produced using the JDE) to capture the design. The Prometheus overview diagram notation (as implemented in the JDE) is also used within AOS on a range of projects.

The methodology has also been taught to undergraduate students as a class. The class spends roughly half of the semester covering the methodology and the other half introducing the JACK agent programming language and platform. The students were able to design and implement reasonable agent systems in a single semester.

⁶The company that produces the JACKTM platform, <http://www.agent-software.com>

Finally, we have on two occasions given undergraduate students materials on Prometheus (tutorial notes and papers) and, with intentionally limited guidance, had them design (and in one case also implement) an agent system. During the Christmas 2002/2003 vacation a second year student was given a description of the methodology and a description of an agent application (in the area of tourism) and asked to design and build a system. Although there was not enough time to build the system (a considerable amount of time was spent in developing requirements based on an available database of tourist information), the student did produce a detailed design in 8 weeks. During the Christmas 2001/2002 vacation a (different) second year student was given a description of the methodology and a description of an agent application (in the area of Holonic Manufacturing) and asked to build a system. With only (intentionally) limited support, the student was able to design and implement an agent system to perform a Holonic Manufacturing simulation in a period of 8 weeks. This was in marked contrast to projects in the late 1990s where students struggled and required large amounts of help, usually ending up with poorly designed agent systems. The feedback from these undergraduate students was valuable in improving the methodology. The two primary issues identified by the students were the need for tool support and the need to simplify the concepts (Prometheus previously had percepts, events, incidents, messages and triggers). Both issues have since been addressed.

7. Related Work

Naturally Prometheus has some similarities to other AOSE methodologies as well as to object-oriented approaches, in particular UML. We review briefly some of these similarities and differences.

7.1 Agent-Oriented Software Engineering

There is currently a large amount of work being done in AOSE methodologies, e.g., (Brazier et al., 1997; Bresciani et al., 2002; Burmeister, 1996; Burafato and Cossentino, 2002; Bush et al., 2001; Caire et al., 2001b; Collinot et al., 1996; Cossentino and Potts, 2002; Debenham and Henderson-Sellers, 2002; DeLoach et al., 2001; Drogoul and Zucker, 1998; Elammari and Lalonde, 1999; Glaser, 1996; Iglesias et al., 1998a; Kendall et al., 1995; Kinny et al., 1996; Lind, 2000; Odell et al., 2000; Shehory and Sturm, 2001; Varga et al., 1994), and, of course, other chapters in this book.

The Gaia methodology has, like Prometheus, been developed over a number of years by people experienced in building agent systems. However, we found that the lack of a detailed design process – intentionally absent due to a desire for generality – meant that it did not provide sufficient support for the needs of those we were working with. There are similarities between Prometheus and

Gaia for specification and architectural design. Our agent acquaintance diagrams are essentially the same as those used by Gaia, and the roles of Gaia are similar in concept to functionalities in Prometheus, although there are slightly different things which are considered.

The Tropos methodology (Bresciani et al., 2002) covers early requirements to detailed design. Its detailed design is oriented very specifically towards JACK as an implementation platform. Compared with Prometheus, Tropos provides an early requirements phase, which Prometheus does not (although, it would certainly be possible to adapt Tropos' early requirements phase for use in Prometheus). Prometheus provides a more detailed process – particularly in the architectural design phase. Prometheus also provides tool support and cross checking; tool support for Tropos is only in the form of a diagram editor⁷, rather than the consistency checking and automatic generation of some parts of the design that is part of PDT.

The MaSE methodology (DeLoach, 2001) (see chapter 6) is one of the few methodologies that has significant tool support. However, MaSE is unsuitable for our purposes since it views agents “... *merely as a convenient abstraction, which may or may not possess intelligence*” (DeLoach, 2001). Thus, MaSE (intentionally) does not support the construction of plan-based agents that are able to provide a flexible mix of reactive and pro-active behaviour.

The MESSAGE methodology (Caire et al., 2001b) (see chapter 9) extends UML to provide rich models for analysis and design. However, there is less detail on detailed design and implementation.

In addition to general purpose methodologies there are also methodologies such as ADELFE (see chapter 8) and SADDE (see chapter 10) which focus on particular application domains or aspects of design. For example, ADELFE extends RUP and UML with activities that support developing complex systems with emergent behaviour. ADELFE has some elements in common with Prometheus: for example characterising the environment, and identifying agent types. However there are also differences: for example goals do not appear to play a significant role in ADELFE.

Because the field is still young, none of the available methodologies can claim extensive use well beyond the group which has developed them. However the widespread development and use of these many new methodologies clearly indicates a need that is starting to be met, to provide more specific design methodologies for building agent systems.

Given the large number of agent-oriented methodologies that have been proposed, there is a growing need for comparisons between methodologies. (Dam and Winikoff, 2003) compare MaSE, Prometheus and Tropos using a feature-

⁷Conversation with Anna Perini at AAMAS in July, 2002.

based approach where the assessment of each methodology against the criteria is validated using a survey of the developers of the methodology (and of students). (Dam, 2003) extends this to include MESSAGE and Gaia and also performs a comparative analysis of the models and processes of each of the methodologies. Other comparisons between agent-oriented methodologies include (Cernuzzi and Rossi, 2002; Shehory and Sturm, 2001; Sturm and Shehory, 2003) and chapter 7.

7.2 Object-Oriented Software Engineering

Some approaches to developing agent-oriented systems are based on taking UML⁸ and extending or modifying it, as is done by (Odell et al., 2000) as well as others with a slightly different approach, e.g., (Papasimeon and Heinze, 2001). This approach is sometimes justified by the argument that agents are a special case of active objects.

Although agents can in some ways be seen as a specialised type of object, we believe that it is important to focus on such concepts as goals, plans and descriptions of situations. This is better supported by a more specialised methodology, borrowing and drawing from UML as appropriate. In our experience, just extending UML does not provide sufficient assistance to start thinking in a different paradigm.

There are significant differences between agent-oriented design and Object-Oriented (OO) design. These differences include the provision of a process for determining the agent types in the system; treating messages as entities in their own right, not just as labels on arcs; distinguishing percepts and actions from messages, and looking explicitly at percept processing; distinguishing beliefs from agents (in OO both are passive objects); the identification of agent life-cycle issues; the use of protocols to capture the dynamics of agent interaction; and the use of goals.

Although there are clear differences between Prometheus and OO methodologies, there are also commonalities. Although we do not believe that current OO methodologies are sufficient, we certainly *do* believe that they are relevant – agents are software (Wooldridge and Jennings, 1998), and indeed, many aspects of the Prometheus methodology have been based on OO methods and notations. For example, the scenarios are adapted from OO use-case scenarios; interaction diagrams are used as-is; AUML (itself an extension of UML) is used as-is, and Prometheus follows the RUP approach to applying an iterative process over clearly delineated phases.

⁸Strictly speaking UML is not a methodology but rather a notation. However it is often coupled, either explicitly or implicitly with a methodology such as the Rational Unified Process (RUP).

In the longer term we see integrating agent methodologies with OO methodologies (and specifically with UML, since it is the de-facto standard notation) as important steps in making agent methodologies accessible to developers. In this respect the work of (Papasimeon and Heinze, 2001) and (Wagner, 2002) is valuable. However, we believe that given the current state-of-the-art, where the concepts and notations for designing agent systems are not agreed upon, it is best to consider possibilities without the world-view suggested by OO.

8. Future Work

Currently we are working on defining in more detail the processes and techniques associated with goals (in the system specification phase), and how these are systematically propagated through the design to the individual plans of an agent. We are also working on processes and techniques that are used to proceed from use case scenarios to interaction protocols. Further tool development is also ongoing.

In the longer term we plan to extend Prometheus with better support for early requirements as well as for implementation, testing and debugging. Secondly, we would like to enhance the methodology to provide specific support for the design of *team* based systems, in the sense of (Cohen and Levesque, 1991), and systems that are *open*. The work of (Collinot et al., 1996; Drogoul and Zucker, 1998; Huget, 2002d) will no doubt be relevant to this latter enhancement.

Finally, as the methodology continues to be used by a broader range of software developers we will continue to respond to evaluations and feedback in an effort to support the process of building MAS.

Acknowledgments

We would like to thank Agent Oriented Software and the Australian Research Council. We would also like to thank James Harland, John Thangarajah, David Poutakidis, Anna Edberg and Christian Andersson of RMIT University, and Ralph Rönquist, Andrew Lucas, Andrew Hodgson, Paul Maisano, and Jamie Curmi of Agent Oriented Software, as well as the many students and workshop participants who have provided comments, examples and feedback.

IV

TOOLS AND INFRASTRUCTURES FOR AGENT-ORIENTED SOFTWARE ENGINEERING

Chapter 12

THE AUML APPROACH

Marc-Philippe Huget, James Odell and Bernhard Bauer

Abstract Since the earliest work in multiagent system development, the need has existed to have a methodology and a modeling notation. A recent approach chosen by several authors is based on UML. The main advantage of UML is to provide a recognized notation in software engineering methodology and strong tools for the development. The approach presented here is called Agent UML (AUML) that synthesizes a growing concern for agent-based modeling representations with the increasing acceptance of UML for object-oriented software development. This chapter covers the first phase (from 1999 to 2002) in the development of Agent UML with the sequence and agent class diagrams and describes future directions that follows Agent UML via the standardization at the FIPA.

1. Introduction

Since the earliest work in MAS development, the need has existed to have a methodology and a modeling notation, see the seminal work on Agent-Oriented Programming (Shoham, 1991). Since this first work from Shoham, many methodologies and notations appear for MAS development. Some chose temporal logic like (Fisher and Wooldridge, 1997) for Concurrent METATEM. A recent approach chosen by several authors is based on UML. The main advantage of UML is to provide a recognized notation in software engineering methodology and strong tools for the development. As (Odell et al., 2000) indicated, starting over to develop a new modeling language for agents was neither useful nor productive. Instead, MAS could, in part, benefit from an incremental extension of existing, known and trusted methods. Agent UML (AUML) synthesizes a growing concern for agent-based modeling representations with the increasing acceptance of UML for object-oriented software development.

Agent UML – as an extension of UML – allows developers coming software engineering to move smoothly from software development to agent development. Agent UML avoids abrupt steps by using and extending UML with agent-specific features through stereotypes and profiles. Agent UML was first

introduced in 1999 with the proposal on interaction protocols given in (Bauer, 1999). After a first period from 1999 till 2002 where Agent UML community proposes two specifications (sequence diagrams and agent class diagrams), a growing interest is visible and efforts become to be coordinated in order to define Agent UML based on the UML 2.0 (OMG, 2003b) specification and to standardize it at the FIPA association. This chapter covers the first period with the description of the two specifications and gives possible future directions that the Agent UML community may follow.

Following sections present the purpose of Agent UML in AOSE (see section 2), the current version of Agent UML with the sequence diagram and the agent class diagram (see section 3). The remaining of the chapter describes future directions of work for Agent UML community particularly with the creation of new diagrams, of tools, the definition of semantics for Agent UML and the development and documentation of applications that use Agent UML for the analysis, design, and implementation.

2. Agent UML Purpose

MAS are often characterized as extensions of object-oriented systems. This overly simplified view has often troubled system designers as they try to capture the unique features of MAS systems using OO tools. In response, an agent-based unified modeling language (AUML) is being developed.

Like UML on which it is partially based, the purpose of Agent UML is to offer to developers a notation that it is used to analyze, design, and implement MAS. The key idea of Agent UML is to reuse as much as possible diagrams coming from UML when they fit MAS designers' needs and to extend UML – through its extension abilities (stereotypes, tagged values, constraints) – when agents and objects are different and agents cannot be represented by UML diagrams. Such an example of extension is in Agent UML sequence diagrams with the three connectors AND, OR and XOR. The use of Agent UML in the documentation is clearly visible in the FIPA Interaction specification (see <http://www.fipa.org/repository/ips.php3>) since all the interaction protocols are documented with a diagram expressed in Agent UML.

It should be noted, however, that instead of reliance on the OMG's UML, we intend to reuse of UML wherever it makes sense. In other words, AUML should not be restricted to just UML – only want to capitalize on UML where appropriate. The general philosophy, then, is: when it makes sense to reuse portions of UML, then do so; when it does not make sense to use UML, use something else or create something new.

3. Current Work in Agent UML

The initial work on Agent UML focuses on the agent interaction. (Odell et al., 2001) define sequence diagrams (also called in some papers protocol diagrams) to this purpose. Agent UML sequence diagrams became *de jure* the notation to represent FIPA Interaction protocols as sketched in FIPA specifications. This first work on sequence diagrams was quickly followed by a proposal for the representation of agents and agent architectures through the agent class diagrams proposed in (Bauer, 2001) and refined in (Huget, 2002a).

3.1 Sequence Diagrams

Agent UML Sequence Diagrams were initially one of the most used diagrams because they were adopted by FIPA to express agent interaction protocols as sketched in FIPA specifications. Sequence diagrams are defined as follows in UML: “A diagram that shows object interactions arranged in time sequence. In particular, it shows the objects participating in the interaction and the sequence of messages exchanged. Unlike a collaboration diagram, a sequence diagram includes time sequences but does not include object relationships.” Since Agent UML considers agents and not objects, one must read agents instead of objects in the previous definition. Sequence diagrams in MAS are diagrams which express the exchange of messages through protocols.

Sequence diagrams have two dimensions: (i) the vertical dimension represents time; and (ii) the horizontal dimension represents different instances or roles. Messages in sequence diagrams are ordered according to a time axis. This time axis is usually not rendered on diagrams but it goes according to the vertical dimension from top to bottom. So, a message defined higher than a second one is sent before. Sequence diagrams do not use sequence numbers as collaboration diagrams to represent the message ordering. Message ordering is performed by the time axis.

Several basic components are used within sequence diagrams:

- 1 Agents and agent roles;
- 2 Agent lifelines and threads of interaction;
- 3 Connectors;
- 4 Messages;
- 5 Conditions on messages;
- 6 Multiplicity;
- 7 Types of message delivery;

- 8 Nested and interleaved messages; and
- 9 Protocol templates.

Following sections present these features.

Agents and Agent Roles. Agents can perform various roles within one interaction protocol. For instance, in an auction between an airline and potential ticket buyers, the airline has the role of a seller and the participants have the role of buyers. But at the same time, a buyer in this auction can act as a seller in another auction. UML precises that a role is the behavior of an entity participating in a particular context. One can also add that a role is a specific set of behaviors, properties, interfaces and service descriptions which allow to distinguish a particular role from another one.

Several terms are used in UML for representing the role classification: a *static classification* means that an agent has an unique role during all the execution. A *multiple classification* corresponds for agents to have different roles during the execution. For instance, it is the case for the previous example where buyers can act, in parallel, as buyers and as sellers. Finally, a *dynamic classification* corresponds to the ability to change from one role to another one during the execution. All these classifications appear when designing sequence diagrams.

A protocol can be defined at the level of concrete agent instances or for a set of agents satisfying a distinguished role and/or class. An agent satisfying a distinguished agent role and class is called *agent of a given agent role and class*, respectively. The general form of describing agent roles in Agent UML is:

instance-1, ..., instance-n / role-1, ..., role-m : class

denoting a distinguished set of agent instances *instance-1, ..., instance-n* satisfying the agent roles *role-1, ..., role-m* with $n, m \geq 0$ and *class* it belongs to. Instances, roles or class can be omitted, in the case that the instances are omitted, the roles and class are not underlined.

Instances, roles and class are textual strings. The class must correspond to an agent class. Roles are rendered within boxes on sequence diagrams. These boxes are places at the top of the diagram.

In Figure 12.1 the auctioneer is a concrete instance of an agent named *UML-Airlines* playing the role of an *Auctioneer* being of class *Seller*. The participants of the auctions are agents of role *AuctionParticipants* which are familiar with auctions and of class *Consumer*. Further information of the elements of this figure are given all along the chapter.

Agent Lifelines and Threads of Interaction. The agent lifeline in sequence diagrams defines the time period during which an agent exists, repre-

Table 12.1. Different combinations for agent names

syntax	explanation
:C	un-named Agent originating from the Class C
/R	un-named Agent playing the Role R
/R:C	un-named Agent originating from the Class C playing the Role R
A/R	an Agent named A playing the Role R
A:C	an Agent named A originating from the Class C
A/R:C	an Agent named A originating from the Class C playing the Role R
A	an Agent named A

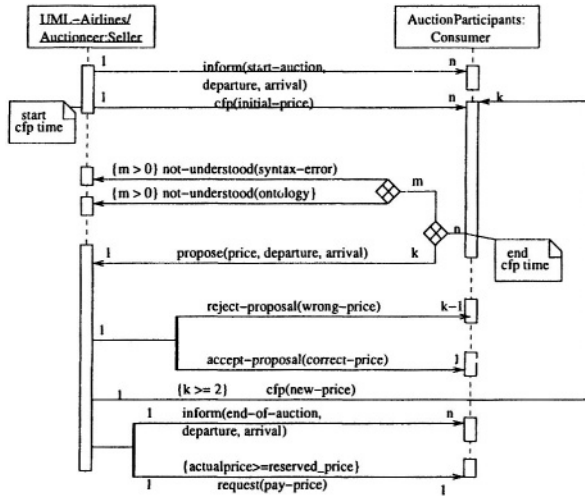


Figure 12.1. English Auction protocol for surplus flight tickets

sented by vertical dashed lines. When a lifeline is created for a role, this role becomes active in the interaction. This lifeline is present as long as the role is active in the interaction.

Added to lifelines, there are threads of interaction – UML uses term *focus of control*. The thread of interaction shows the period during which an agent is performing some tasks as a reaction to an incoming message. It only represents the duration of an action, but not the control relationship between the sender of the message and its receiver. A thread of interaction is rendered as a tall thin rectangle superposed on lifelines. If a metric is defined for the time axis, the

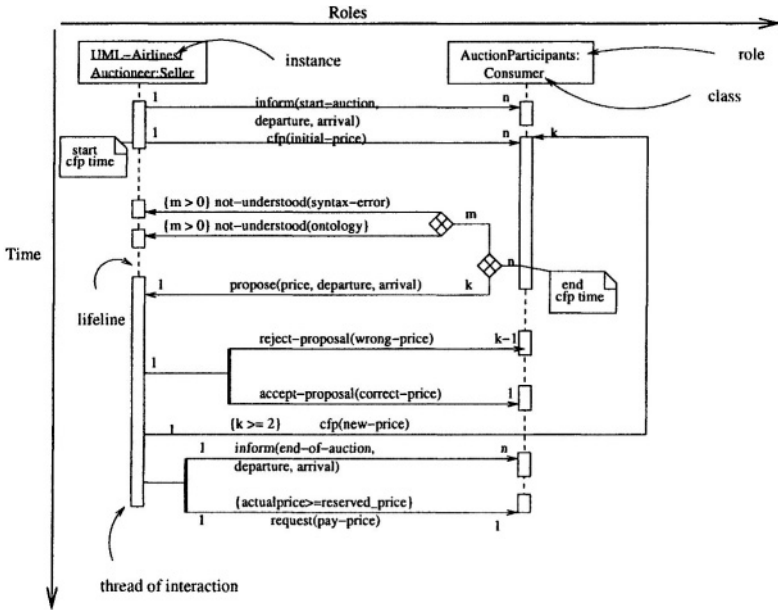


Figure 12.2. Sequence Diagram

thread of interaction corresponds exactly to the time needed to perform actions triggered by the incoming messages.

In this way, it is possible to follow the path in the interaction from the initial interaction state to the final interaction state. Actually, one has just to read the first message on the sequence diagram (messages are ordered from top to bottom), follows the transitions and uses the thread of interaction in order to find the next message. An example is shown on Figure 12.2.

The first message is the *inform* message. This message arrives to the *AuctionParticipants* but no outgoing messages are available for the thread of interaction. In fact, the next message belongs to the thread of the *Auctioneer*. This is the *cfp* message. This time, there are outgoing messages for this message. Several message traces are defined from this point. *AuctionParticipant* can answer either with *not-understood* or *propose* (see Figure 12.2).

Connectors. The lifelines may be split in order to demonstrate two kinds of behaviors: parallelism and decisions. Three connectors are supplied for these features (shown on Figure 12.3). The connector AND is rendered as a thick vertical line as shown on Figure 12.3a. It means that messages have to be sent concurrently. On Figure 12.3a, *CA-1*, *CA-2*, *CA-3* are sent in parallel. The connectors OR is rendered as a diamond as shown on Figure 12.3b and

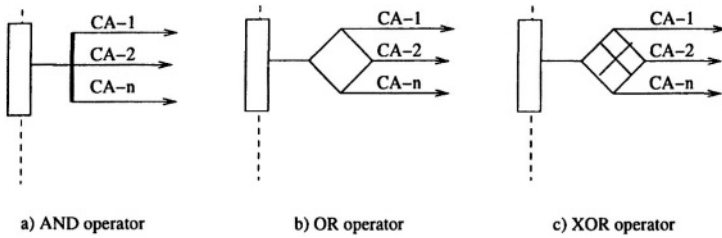


Figure 12.3. Agent UML Connectors

XOR is rendered as a diamond and a cross within it as shown on Figure 12.3c. They mean that a decision between several messages has to be done. When considering the connector OR, zero or several messages is chosen: a subset of the set $\{CA-1, CA-2, CA-3\}$. In the case of several messages are selected, the messages are sent in parallel. The connector XOR also represents a decision but in this case, one and only one message is chosen, it is either $CA-1$ or $CA-2$ or $CA-3$.

Messages. Messages in sequence diagrams are sent from a sender role to receiver roles. A message is rendered as a directed solid line. The arrowhead points out the receiver role of this message. Messages are adorned with several information as shown on Figure 12.4:

- 1 A textual string which is the message. If designers use FIPA ACL, messages could be the communicative act and the content.
- 2 Conditions which must be satisfied in order to enable an associated transition to fire. Conditions may be written as free-form text. Formal conditions are written with OCL as depicted in (OMG, 2003c). Conditions are nested by curly braces.
- 3 Multiplicity with two numbers placed near the sender role and the receiver role. These numbers correspond to the number of messages that are sent by the sender role and the number of instances in the receiver role that receive each message.

Several examples are given on Figure 12.1, for instance, the first message from the role *Auctioneer* to the role *AuctionParticipants* where the message is *inform(start-auction, departure, arrival)*.

Conditions. The connectors OR and XOR are examples where a decision has to be performed to choose the next messages. A casual solution for tackling the non determinism of the sequence diagrams is to use conditions. As a consequence, a message can be sent if and only if the conditions attached to this

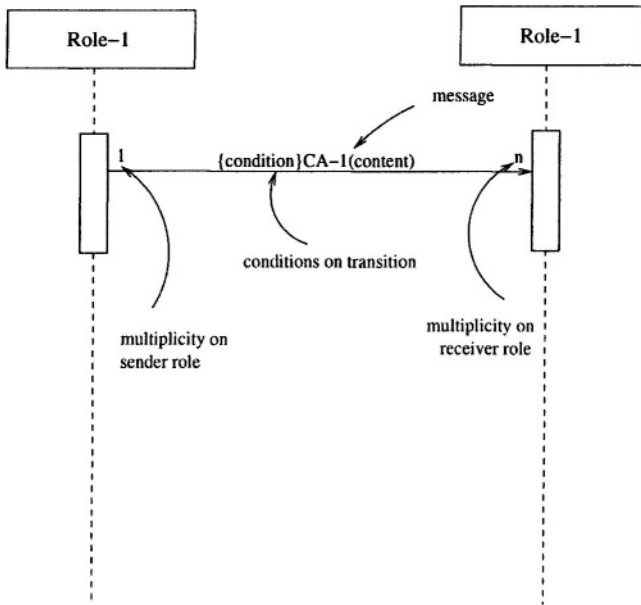


Figure 12.4. Messages in Agent UML

message are satisfied. Conditions on sequence diagrams are rendered a textual string nested by curly braces as shown in Figure 12.1 for the *not-understood* messages. The textual string can be written according to a free format or designers can use OCL (*Object Constraint Language*) defined for this purpose as depicted in (OMG, 2003c). Conditions are written just before the message on sequence diagrams as shown on Figure 12.4.

Multiplicity. Sequence diagrams represent agents either by their instances or by their role in the protocol. When using roles, it is interesting to know the number of agents involved in both the sender role and the receiver role. The cardinality for sender and receiver roles are given by the multiplicity. It is for instance the case on Figure 12.1 for the *propose* message where k *AuctionParticipants* send a *propose* to the *Auctioneer*. Sometimes, it is not possible to give the exact number of messages that are received. For instance, it is the case for the English Auction protocol since it is not possible to know how many *AuctionParticipants* will answer to the bid.

Multiplicity is adorned at both end of the transition. For instance, if the message is sent by one agent role to n other agents, the value 1 is written near the lifeline of the sender role and n is written near the lifeline of the receiver

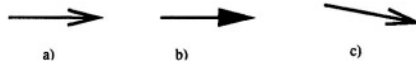


Figure 12.5. Types of Message Delivery in Agent UML

role. Obviously, if several instances in sender role send the message, designers have to write the number corresponding to the number of agents instead of 1.

Multiplicity can be represented by a range of value: *0..1* (zero or one), *0..** (many) or *1..** (one or more). It is also possible to write several ranges of values such as *1..4*, *6..** which means at least one message and not five messages.

Several examples are given on Figure 12.1. For instance, for the *cfp* message, the multiplicity is *1..n*. It means that the message is sent to *n Auction-Participants*. The star could be used to denote that zero or more copies of this message are sent.

Types of Message Delivery. Messages in interaction are usually sent asynchronously (the symbol with a stick arrowhead as shown on the Figure 12.5a). It shows the sending of the message without yielding control. It is also possible to send messages synchronously (the symbol with a filled solid arrowhead as shown on the Figure 12.5b). It shows the yielding of the thread of control (wait semantics), i.e., the agent role waits until an answer message is received and nothing else can be processed. Normally, messages are drawn horizontally. This indicates the duration required to send the message is “atomic,” i.e., it is brief compared to the granularity of the interaction and that nothing else “happen” during the message transmission. If the messages require some time to arrive, for instance for mobile communication, during which something else can occur. The message is shown on Figure 12.5c.

Nested and Interleaved Protocols. Because protocols can be codified as recognizable patterns of agent interaction, they become reusable modules of processing that can be treated as first-class notions. For example, Figure 12.6 depicts two kinds of protocol patterns. The left part defines a nested protocol, i.e., a protocol within another protocol, and the right part defines an interleaved protocol, e.g., if the participant of the auction requests some information about his/her bank account before bidding. Additionally nested protocols are used for the definition of repetition of a nested protocol according to conditions. The semantics of a nested protocol is the semantics of the protocol. If the nested protocol is marked with some conditions then the semantics of the nested protocol is the semantics of the protocol under the assumption that the conditions evaluate to true, otherwise the semantics is the semantics of an empty protocol, i.e., nothing is specified.

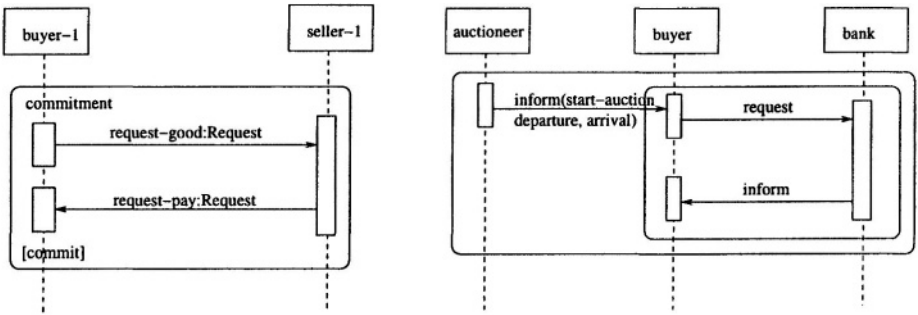


Figure 12.6. Nested Protocol and Interleaved Protocol

Protocol Templates. The purpose of a protocol template is to create a reusable pattern for useful protocol instances. For example, Figure 12.7 shows a template for the FIPA-English-Auction Protocol from Figure 12.1. It introduces two new concepts represented at the top of the sequence diagrams. First, the protocol as a whole is treated as an entity in its own right. The protocol can be treated as a pattern that can be customized for other problem domains. The dashed box at the upper right-hand corner declares this pattern as a *template* specification that identifies unbound entities (formal parameters) within the package that need to be bound by actual parameters when instantiating package. A parameterized protocol is not a directly-usable protocol because it has unbound parameters. Its parameters must be bound to actual values to create a bound form that is a protocol. Communicative acts in the formal parameter list can be marked with an asterisk, denoting kinds of messages which can alternatively be sent in this context. This template can be instantiated for a special purpose as shown in Figure 12.1. Figure 12.8 applies the FIPA English Auction Protocol to a particular scenario involving a specific auctioneer *UML-Airlines* of role *Auctioneer* and class *Seller* and *AuctionParticipants* of class *Consumer*. Finally, a specific deadline has been supplied for a response by the seller.

3.2 Agent Class Diagrams

Class diagram in UML shows a set of classes, interfaces, and collaborations and their relationships. Class diagrams are the most common diagram found in modeling object-oriented systems. Class diagrams are used to illustrate the static design view of a system. Generally, class diagrams contain three information: class attributes, class operations, and the relationships between classes. It is also possible to insert the class interfaces and other compart-

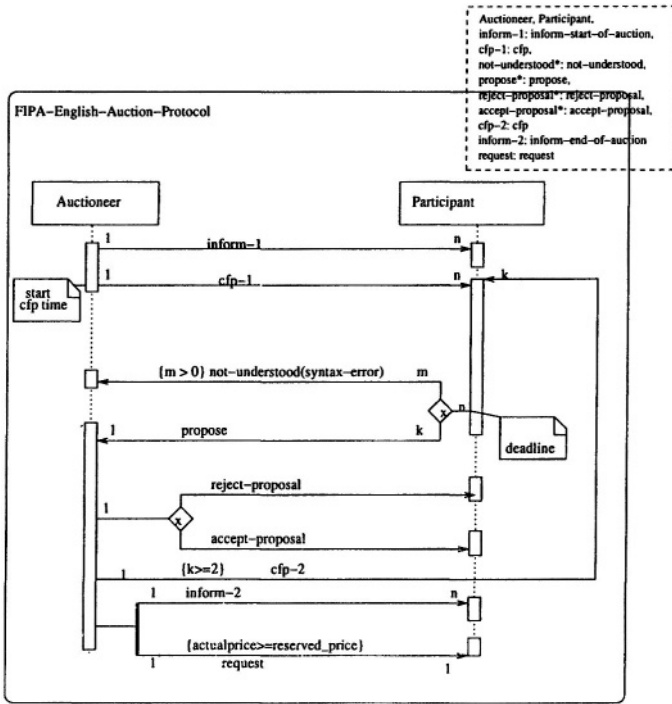


Figure 12.7. Nested Protocol and Interleaved Protocol

FIPA-English-Auction-Protocol

UML-Airlines / Auctioneer : Seller, AuctionParticipants : Consumer

start cfp time + 1 min

inform(start auction, departure, arrival),

cfp(initial-price),

not-understood(syntax error),

not-understood(ontology),

propose(pay-price),

reject-proposal(wrong-price),

accept-proposal(correct-price),

cfp(increased-price),

inform(end-of-auction),

request(pay-price, fetch-car)

Figure 12.8. Instantiation of a Protocol Template

ments in the class view such as responsibilities or exceptions. Due to many differences between agents and objects, see (Odell, 2002) for a description, class diagrams are modified deeply in order to encompass agent features such as knowledge, plans or protocols used. For pointing out the differences with class diagrams in UML, class diagrams in Agent UML are called agent class diagram as depicted in (Bauer, 2001).

UML distinguishes different specification levels, namely the *conceptual*, the *specification* and the *implementation level*. Behind these levels are the notion of abstraction. The abstraction allows designers to focus on some relevant details while ignoring others. Designers can define as many views as they want.

On the *conceptual level*, designers provide an overall view of a system: the different agent classes and their relationships without considering what could be the elements within classes. This level is particularly fitted for a description of the architecture of the system.

On the *specification level* or interface level, an agent class is blueprint for instances of agents. But only the interfaces are described and not the implementation, i.e., the agent head automata describing the behavior of the agent according to incoming messages is missing. Only the internal states and the interface, i.e., the communicative acts supported by the agent, is defined.

The *implementation level* or code level is the most detailed description of a system, showing how instances of agents are working together and how the implementation of a class of agents looks like. On this level the agent head automata has to be defined, too.

UML class diagrams are not defined here. Detailed description of UML class diagrams are given in (Booch et al., 1999).

Agent class diagrams as defined in (Bauer, 2001) contain several elements:

- Agent name;
- State description;
- Actions;
- Methods;
- Capabilities, service description, supported protocols;
- Organization belonging; and
- Agent head automata.

Agent Name. Agents are different from objects so it is necessary to make distinction when agents and objects are both used on the same diagram. This

is the case when agents are defined as a set of objects or when they use objects to perform their tasks. The stereotype «agent» prefixes agent name.

Three information may be supplied for an agent name: instance, role and class. Instances, roles and classes correspond to three levels when considering agents. Class is the most general one. A class is a set of objects that share the same set of attributes, operations and relationships and have the same semantics in UML. The definition is extended for Agent UML in order to take into consideration plans, knowledge or protocols. A role is the behavior associated to an entity into a particular context. For instance, two roles are usually described in auctions: seller and bidders. The role defines how agents react to events. Two different roles have two different behaviors. Instances are the most accurate information. Instances give the name of each agent involved. Instances are unique, i.e., if one has two instances A and B, it is not possible to use the instance A instead of the instance B and vice-versa. The general form of describing agent roles in Agent UML is:

instance-1, ..., instance-n / role-1, ..., role-m : class

denoting a distinguished set of agent instances *instance-1, ..., instance-n* satisfying the agent roles *role-1, ..., role-m* with $n, m \geq 0$ and *class* it belongs to. Instances, roles or class can be omitted, in the case that the instances are omitted, the roles and class are not underlined.

State Description.

The state description looks similar to the attributes compartment in class diagrams except that we introduce a distinguished class *wff* for *well formed formula* for all kinds of logical descriptions of the state, independent of the underlying logic. With this extension we have the possibility to define as well BDI agents.

In the case of BDI semantics, one can define four instance types, named *beliefs*, *desires*, *intentions* and *goals*, each of type *wff*. These fields can be initialized with the initial state of BDI agents.

Attributes follow the same construction as for attributes in UML:

[visibility] name [multiplicity] [:type]
[= initial-value] [property-string]

Visibility defines how an attribute can be seen and used by others. Three cases are available: *public*, *private* and *protected*. The public visibility means that other classes can access this attribute. Public visibility is denoted by the symbol '+'. The private visibility means that only attributes and operations in the same class of this attribute can access it. Private visibility is denoted by the symbol '-'. The protected visibility means that the class itself and all descendant of this class can access it. Protected visibility is denoted by the symbol '#'.

Name is the name of the attribute. It is a textual string. The name must be unique within the class.

Multiplicity is used when it is necessary to represent several copies of the same attribute. There are two methods for describing multiplicity: either with a number or with a range of values. Designers use numbers or ranges of values whether they know exactly the number of copies or not.

Type represents the type of the attribute.

Initial-value describes the initial value of this attribute.

Property-string defines how attributes can be used: *changeable* is the default value and means that it is possible to update the value of this attribute, *add-only* is used for lists and means that only the insertion is possible, it is then not possible to update or to delete values in the list, *frozen* corresponds to constants, the value of the attribute cannot be modified, *persistent* denotes that the value is persistent.

Actions. Two kinds of actions can be specified for an agent: pro-active actions (denoted by the stereotype «pro-active») that are triggered by the agent itself, e.g., using a timer, or a special state is reached. It is tested on state changes of the agent (e.g., timer, sensor input) if the pre-condition of the actions evaluates to true. Re-active actions (denoted by the stereotype «re-active») are triggered when receiving some message from another agent.

In its full form, the syntax of an action is:

```
[visibility] [pre-conditions] name [(parameter-list)]
                               [post-conditions]
```

Visibility has the same definition as the one shown before. *Name* is a textual string.

Parameter-list contains both the name of the parameter and its type.

Pre-conditions are constraints that must be true when an action is invoked.

Post-conditions are constraints that must be true at the completion of an action.

Pre-conditions and post-conditions may be written as a free-form text or as an OCL expression as expressed in (OMG, 2003c).

Methods. Methods are defined like operations in UML. An operation is the implementation of a service that can be requested from any object of the class to affect behavior. In other words, an operation is an abstraction of something you can do to an object and that is shared by all objects of that class.

In its full form, the syntax of a method is:

```
[visibility] [pre-conditions] name [(parameter-list)]
```

```
[ : return-type ] [ post-conditions ] [ property-string ]
```

Visibility has the same definition as the one before.

Name is a textual string.

Parameter-list is defined as follows:

```
[ direction ] name : type [= default-value]
```

Direction may be any of the following values:

- *in*: an input parameter; may not be modified;
- *out*: an output parameter; may be modified to communicate information to the caller; and
- *inout*: an input parameter; may be modified.

Type and *return-type* represent the type of the action and of the parameter respectively. *Default-value* describes the default value of the parameter.

Pre-conditions are constraints that must be true when an action is invoked.

Post-conditions are constraints that must be true at the completion of an action.

Pre-conditions and post-conditions may be written as a free-form text or as an OCL expression as expressed in (OMG, 2003c).

Capabilities, Service Description and Supported Protocols. Capabilities describe what agents can do. UML does not supply capabilities in its diagrams but capabilities is close to the notion of responsibilities in UML. These responsibilities are represented as a free-form text.

Service description has to be linked to interface in UML. An interface in UML is a collection of operation that are used to specify a service of a class or a component. The operations in the interface do not have attributes. These operations are considered as entry points for the class linked to these operations. One clear advantage of interfaces is that designers do not have to modify classes linked to an interface as soon as this interface is modified. Classes remain the same as long as interfaces keep the same name. Service descriptions are rendered as an interface name and a “lollipop” linked to the class. This rendering does not describe the operations of the service description. Service descriptions are represented with their operations as a class but prefixed by the keyword «service».

Supported protocols are described as a list. Supported protocols are adorned with the roles played by the agent in these protocols.

Group Representation. Agents do not act solely. They belong to at least one group and play one or several roles in this group. The compartment *organization* gives the different groups in which the agent evolves, which roles it plays and under which constraints, it can evolve in these groups. The syntax for this information is the following:

```
[constraint] organization : role
```

Constraints are written as a free-form text or as an OCL expression. Constraints must be satisfied if agents want to belong to this group.

Role matches the roles defined for this agent in the agent name.

Agent Head Automata. The agent head automata defines the behavior of an agent's head. Agents are composed of three parts: communicator, head and body.

The *agent communicator* is responsible for the physical communication of the agent. The main functionality of the agent is implemented in the *agent body*. This can be, e.g., an existing legacy software which is coupled to the MAS using wrapper mechanisms.

The *agent's head* is the "switch-gear" of the agent. Its behavior has to be specified with the agent head automata. Especially, this automata related to the incoming messages with the internal state, actions, methods and the outgoing messages, called the reactive behaviors of the agent. Moreover, it defines the pro-active behaviors of an agent, i.e., it automatically triggers different actions, methods and state-changes depending on the internal state of the agent. An example of pro-active behavior is to do some action at a specific time, e.g., an agent migrates at pre-defined times from one machine to another one, or it is the result of some request-when communicative acts.

4. Future Directions in Agent UML

2003 will be the year of deep modifications in Agent UML thanks to the issue of the new UML 2.0 defined in (OMG, 2003b) and the growing interest in Agent UML. Most of the Agent UML work is now being performed within FIPA's Modeling Technical Committee (TC). An important goal of the Modeling TC is to be domain independent. Currently, the TC is examining those area where its members have expertise: Service-Oriented Architecture (SOA), Business Process Management (BPM), simulation, real-time, AOSE, robotics, information systems. Other areas will be examined over time as additional expertise becomes available.

Two diagrams will initially be part of the FIPA specification. The first is the interaction diagram. This diagram consists of sequence diagrams, communication diagrams (previously called collaboration diagrams in UML 1) and the interaction overview diagram. The second is the agent class diagram, which

will be inspired by the UML class diagram and extended to express agent-based notions. One can think that this emerging effort around Agent UML is the first step to a better Agent UML comprising several new diagrams and tools to exploit them. This section describes possible future directions of work for Agent UML community.

4.1 Diagrams

Section 3 describes the current version of Agent UML at time of writing this chapter where two diagrams are considered: *sequence diagrams* to represent interaction between agents and *agent class diagrams* to represent agents and agent architectures. From the first work in 1999 till 2002, this specification remains as proposed but the growing interest around Agent UML mostly the last two years gave birth to an effort to update this specification and add new diagrams, some derived from UML diagrams, some specifically tailored for agent needs. The year 2003 is the year of deep modifications in Agent UML partly due to the issue of a new version of UML called UML 2.0 defined in (OMG, 2003b). This new version of UML enhances the dynamic models and particularly the *interaction diagrams* that merge several diagrams and especially, sequence diagrams. These interaction diagrams now more clearly outline the different traces in an interaction through the CombinedFragments described in (OMG, 2003b). As a consequence, Agent UML sequence diagrams are hence called interaction diagrams. The current version of Agent UML interaction diagrams can be found on Agent UML Web site <http://www.auml.org>. Since this is an ongoing work, we here just depict the main modifications in Agent UML interaction diagrams.

Sequence Diagrams The interaction is still enclosed in a frame but this time, a label is added with the protocol name prefixed by the keyword *sd* as shown on Figure 12.9. Moreover, template protocols are more clearly distinguishable since the stereotype `<template>` is written in the label whereas, in the current version, designers have to figure out that this is a template since there is a dotted box overlapping the diagram in the upper right corner.

Lifelines are deeply modified: in the current version, agents are depicted through their identity and their roles in the interaction, a role can be suffixed by the class of the agent. In the new version, agents are depicted through their identity and their roles but groups replace classes. Groups are attached to roles. One found important to represent that a role can have a different behaviour if it is involved in two different groups. Role cardinality is also included in the box on top of the lifeline. Finally, the main difference about the lifeline is the ability for agents to change of roles or to add and delete roles during the interaction, that is to say an agent winning an auction can move to the role *winner* if needed. Messages are slightly modified to take into consideration when mes-

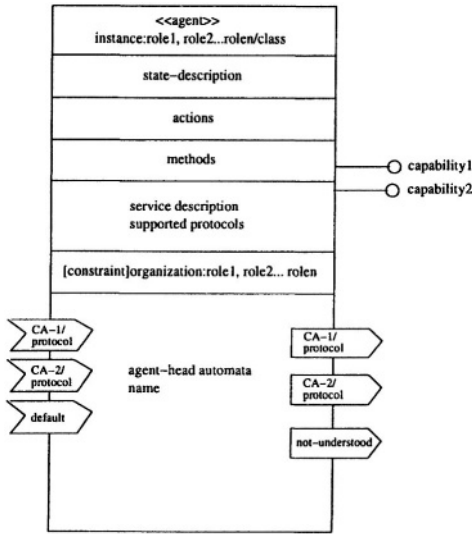


Figure 12.9. Contract Net in AUML 2

sages are sent to a specific agent or when messages are sent to the same lifeline (the sender is whether receiver or not). Timing constraints are now added to interaction diagrams allowing designers to represent deadlines like in Contract Net depicted in (Davis and Smith, 1983). Finally, the main difference between the current version and the new version remains in the splitting/merging path feature. In the current version of sequence diagrams, only three connectors are available: AND (for parallel sending), OR (1 or several messages are sent) and XOR (1 and only 1 is sent). This choice is rather restricted, the new version of Agent UML interaction diagrams also considers the notion of loop, break in the interaction to name a few, and the ability to combine different operators.

Class Diagrams A second effort is processed around agent class diagrams in order to represent a wide range of agents and partly due to some critics about the cumbersome of the notation and some inefficient choices as showed in (Huget, 2002a). At time of writing, the agent class diagram specification is not enough advanced to be presented here. The draft version of the new Agent UML class diagrams can be found on Agent UML Web site as well.

It is worth noticing that these two specifications are about to be standardized at FIPA.

Various Diagrams These Interaction and Class diagrams are the first ones that will be standardized at the FIPA. Extensions to the other UML represen-

tations are planned. Currently, these include: packages, templates, activity diagrams, class diagrams, deployment diagrams, and state machines. As mentioned earlier, the FIPA Modeling TC activities are not limited to UML for their inspiration: we intend to reuse of UML wherever it makes sense. There is already some research being conducted to model which UML does not currently address. For instance, roles and groups are now playing a more important role in interaction diagrams. The need for a role diagram that expresses how agents assume and change roles is vital. A second kind of diagram can emerge to deal with goals and planning. These are just two examples among others of possible diagrams in Agent UML. Perhaps all the UML diagrams will be considered and “agentified” if appropriate. The efforts of the FIPA Modeling TC are ongoing and therefore not fully planned. The Modeling TC cannot fully plan, because it is still too early to know exactly what will be required to model agent-based systems. The field of agents and agent-based systems is still in its early stages. As the agent community understands better how to think, communicate, and represent its notions of agents, we will then be develop an richer and more expressive AUML.

4.2 Tools

All specifications are worthless if no tools are provided to help designers representing, exchanging diagrams and generating code from these diagrams. Unfortunately, this point is not really considered for the moment for Agent UML. Two options are possible: (i) updating an existing tool; or (ii) creating from scratch a tool dedicated to Agent UML.

The former option is certainly the less time consuming. Two kinds of tools are considered for this approach: tools that support UML and tools that simply draw diagrams. In the second case, it is straightforward to propose an extension to support Agent UML; some work is done for the moment around Dia (see <http://www.lysator.liu.se/~alla/dia>) and Microsoft Visio. The main drawback of this approach is the absence of integrated environment to generate code and check diagrams. Problems happen with tools that support UML since UML 2.0 is a young specification and tools do not support it for the moment, as a consequence we cannot extend them to support Agent UML. The advantage of such tools is the generation of code and the diagram validation. OpenTool (see <http://www.tni-valiosys.com>) supports UML and Agent UML in its current version.

Creating a tool from scratch to support Agent UML is certainly the most time-consuming task but it offers a tool that perfectly answers to Agent UML needs. Maybe some tools will appear in near future as soon as projects and applications will consider Agent UML as notation.

Our first work will be to extend current tools that support UML 2.0 in order to support Agent UML as well. As soon as Agent UML will become stronger and used, the solution will be to tailor a specific tool that offers diagram modeling, code generation and validation.

4.3 Algorithms

We have already advocated the needs for tools that consider both diagram modeling, code generation and validation. Code generation and validation of such diagrams in the context of MAS have to be defined. Some work already exist for the current version of sequence diagrams, see (Huget, 2002c; Koning and Romero-Hernandez, 2002), but the same kind of effort has to be applied to the new specification of interaction diagrams. Except the work in (Huget, 2002b) where a Java program is generated from a sequence diagram, there is no work around code generation.

Important work has to be performed around agent class diagrams since nothing for code generation and validation exists for the moment.

As soon as the interaction diagram and the agent class diagram specifications will be accepted, we will move part of our effort to provide such algorithms for the validation of these diagrams and for the generation of code.

4.4 Semantics

A critic frequently encountered both for UML and Agent UML is these two notations are not formal. We understand this concern and are conscious that this can cause some misunderstandings when modeling agents or protocols if this interpretation is not clearly defined. We actively work at the definition of a semantics for Agent UML leveraging the critics on formalism in Agent UML.

4.5 Applications

Applications have to be considered as first-class citizens in the near future of Agent UML. Actually, there is no important application of Agent UML for the moment. The design of applications will arise problems and lacks in the specification and will offer tools for the design of Agent UML.

5. Conclusion

MAS are often characterized as extensions of object-oriented systems. This overly simplified view has often troubled system designers as they try to capture the unique features of MAS systems using OO tools. In response, the agent-based unified modeling language – Agent UML – is being developed. Instead of reliance on the OMG's UML, we intend to reuse of UML wherever it makes sense. We do not want to be restricted by UML; we only want to

capitalize on it where we can. The general philosophy, then, is: when it makes sense to reuse portions of UML, then do it; when it does not make sense to use UML, use something else or create something new. Upon closer inspection many of the new breakthroughs of UML 2.0 reflect the requests of the agent community – making AUML less of an extension to UML 2.0 than UML 1.0.

Chapter 13

FIPA-COMPLIANT AGENT INFRASTRUCTURES

Fabio Bellifemine and Agostino Poggi

Abstract This chapter is an introduction to FIPA-compliant agent infrastructures and, in particular, to JADE that is one of the most known and used agent development framework. FIPA is an international non-profit association of companies and organizations sharing the effort of producing specifications for generic agent technologies that can enable end-to-end interoperability between agent systems. JADE is a software environment to build agent systems for the management of networked information resources in compliance with the FIPA specifications.

1. Introduction

In these last years, there has been a lot of activity concerning the standardization of agent-based technologies. In fact, during these years, three important initiatives, KSE (Patil et al., 1992), FIPA (see <http://www.fipa.org>) and OMG (see <http://www.omg.org>), worked with the goal of realizing specifications for agent technology, and, at the end, FIPA defined a set of specifications that are now considered a “de facto” standard for agent interoperability. An important consequence of this result is that “to be FIPA-compliant” became a key feature of any agent based product; therefore, people working on agent development tools and libraries are more and more interested in offering the possibility to realize “FIPA-compliant” agent based products.

This chapter is an introduction on FIPA-compliant agent infrastructure and, in particular, on JADE that is that most known and used agent development framework compliant to FIPA specifications. Next section describes FIPA activities and specifications. Section 3 introduces some of the most important FIPA compliant development tools and libraries. Section 4 presents the main features of JADE and some notes about its use in some international projects. Finally, last section concludes with a brief discussion on FIPA and FIPA-compliant development tools.

2. FIPA

The Foundation for Intelligent Physical Agents (FIPA) is an international non-profit association of companies and organizations sharing the effort to produce specifications for generic agent technologies. FIPA does not promote a technology for just a single application domain but a set of general technologies for different application areas that developers can integrate to make complex systems with a high degree of interoperability.

The FIPA standardization process relies on two main assumptions. The first is that the time required to reach a consensus and to complete the standard should be as short as possible, should not impede progress, but should act as a promoter of stronger industrial commitment in agent technology. The second assumption is that only the external behaviour of system components should be specified, leaving implementation details and internal architectures to platform developers. In fact, the internal architecture of JADE is proprietary even if it complies with the interfaces specified by FIPA.

Based on the set of preliminary specifications released in 1997, at the end of 2002 FIPA released its standard.

The FIPA standard defines the reference model of an agent platform and a set of services that should be provided. The collection of these services, and their standard interfaces, represents the normative rules that allow a society of agents to exist, operate and be managed. The standard identifies the roles of some key agents necessary for managing the platform, and describes the agent management content language and ontology. Two key roles were identified for an agent platform (see Figure 13.1). The Agent Management System (AMS) is the agent that exerts supervisory control over access to and use of the platform; moreover, it is responsible for maintaining a directory of resident agents and for handling their life cycle. The Directory Facilitator (DF) is the agent that passes on yellow page services to the agent platform. Notice that no restriction is given to the actual technology used for platform implementation: an e-mail based platform, a CORBA based one, a Java multi-threaded application, etc. could all be FIPA compliant implementations.

According to FIPA definition, an agent is the fundamental actor in a domain. It is capable of bringing together a number of service capabilities to form a unified and integrated execution model that can include access to external software, human users and communication facilities.

Of course, the specifications also define the Agent Communication Language (ACL). Agent communication is based on message passing, i.e., agents communicate by sending individual messages to each other. The FIPA ACL is

atics of the messages. The specifications of the FIPA ACL and its underlying model of communication is based on three main properties:

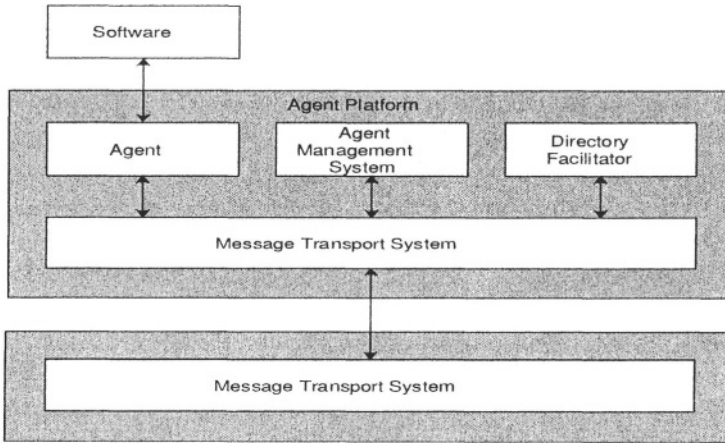


Figure 13.1. FIPA agent platform reference model

- *Agents are active entities, they can say “no,” and they are loosely coupled.* This set of interrelated properties forms the basis of the choice of message-based asynchronous communication between agents as opposed to simple method call: an agent wishing to communicate has just to send a message to a certain destination. This modality of communication, in fact, allows the receiver to select which messages to serve and which to discard, as well as which messages to serve first and which later in time. It also allows the sender to control its thread of execution and not to be blocked until the receiver reads and serves the message. Finally, it also removes any temporal dependency between the sender and the receiver: the receiver might not be available at the time the sender sends the message, or it might even not exist at that time, or, also, it might even be not known by the sender that, instead, defines the receiver intentionally (e.g., the agent interested into “football”) or mediates the communication through a proxy (e.g., propagate this message to all agents in the system X).
- *Agents perform actions and communication is just a type of action.* Making communication at the same level of actions allows an agent, for instance, to reason about a plan that includes both physical actions (e.g., turning on the left) and communicative actions (e.g., asking to open the door). In order to make communication plannable, effects and preconditions of each possible communication needs to be clearly defined.
- *Communication carries a semantics meaning.* When an agent is the object of a communicative action (i.e., when it receives a message), it must be able to properly understand the meaning of that action and, in partic-

ular, why that action has been performed (i.e., the communicative intention of the sender of the message). This property turns into the needs for a universal semantics and the need for a standard, as the one described later.

The FIPA ACL also provides the bases for the specification of interaction protocols, which are common patterns of conversation between agents aimed at specifying high-level tasks, such as delegating a task, negotiating conditions, and some forms of auctions.

The remaining parts of the FIPA specifications deal with other aspects, in particular with agent-software integration, agent mobility, agent security, ontology service, and human-agent communication; however, these specifications are either not complete or have a limited experimentation.

3. FIPA-Compliant Agent Infrastructures

In the last years, FIPA is becoming the “de facto” standard for agent interoperability, therefore, people working on agent development tools and libraries consider to be “FIPA-compliant” a key feature of their products. In fact, the large part of the development systems, that have been realized in the last years or that are under development, are FIPA-compliant, see, e.g., ASL (Kerr et al., 1998), Bee-gent (Kawamura et al., 1999), FIPA-OS (Poslad et al., 2000), Grasshopper (see <http://www.grasshopper.de>), Opal (Purvis et al., 2002), Zeus (Nwana et al., 1999).

ASL is an agent platform that supports the development in C/C++, Java, JESS, CLIPS and Prolog (Kerr et al., 1998). ASL is built in line with the OMG’s CORBA 2.0 specifications. The use of CORBA technology facilitates seamless agent distribution and makes it possible to add the language bindings supported by the CORBA implementations to the platform. Initially, ASL agents communicated through KQML messages, now the platform is FIPA compliant supporting FIPA ACL.

Bee-gent is a software framework to develop agent systems compliant with FIPA specifications produced by Toshiba (Kawamura et al., 1999). Such a framework provides two types of agents: wrapper agents used to agentify existing applications and mediator agents supporting the wrappers coordination by handling all their communications. Agents communicate through XML/ACL messages and mediator agents are mobile agents that can migrate around the network by themselves. Bee-gent also offers: (i) a graphic RAD tool to describe agents through state transition diagrams; (ii) naming/directory facilities to locate agents, databases and applications; (iii) ontology facilities to translate words referring to the same entity; and (iv) security and safety facilities based on digital fingerprint authentication and secret key encryption in order

to prevent the mediator agents from being tampered with, or wiretapped while moving around the network.

FIPA-OS is another software framework to develop agent systems compliant with FIPA specifications that has been created by Nortel Networks (Poslad et al., 2000). Such a framework provides the mandatory components that produce the agent platform of the FIPA reference model (i.e., the AMS, ACC and DF agents, and an internal platform message transport system), an agent shell and a template to produce agents that communicate by taking advantage of FIPA-OS agent platforms.

Grasshopper is a pure Java based mobile agent platform, conforming with existing agent standards, as defined by the OMG - MASIF (Mobile Agent System Interoperability Facility) (Milojicic et al., 1998) and FIPA specifications. Thus Grasshopper is an open platform, enabling maximum interoperability with other mobile and intelligent agent systems, and easy integration of existing and upcoming CORBA and Java services and APIs. The Grasshopper environment comprises several Agencies and a Region Registry, remotely connected via a selectable communication protocol. Several interfaces are specified to enable remote interactions between the distinguished distributed components. The life of an agent system is based on the services offered by a core agency. A core agency provides only those capabilities that are absolutely crucial for the execution of agents. Agents access the core agency for retrieval of information about other agents, agencies or places, or in order to move to another location. Users are able to monitor and control all activities within an agency by accessing the core services, i.e., Communication Service, Registration Service, Transport Service, Security Service, and Management Services. Moreover, Grasshopper provides a graphic user interface for user-friendly access to all of the functions of an agent system.

The Opal architecture for software development is described that supports the use of agent-oriented concepts at multiple levels of abstraction Opal (Purvis et al., 2002). At the lowest level are micro-agents, streamlined agents that can be used for conventional, system-level programming tasks. More sophisticated agents may be constructed by assembling combinations of micro-agents. The architecture consequently supports the systematic use of agent-based notions throughout the software development process. With Opal it is possible to design FIPA-based agent systems and also employ agent-based components for virtually all aspects of a software system, including finer-grained components that are not normally implemented in terms of agent constructs for reasons of efficiency. Opal also supplies an Agent Conversation Manager that incorporates the notion of higher-level "policies" for guiding and constraining agent interactions.

Zeus allows the rapid development of Java agent systems by providing a library of agent components, by supporting a visual environment for capturing

user specifications, an agent building environment that includes an automatic agent code generator and a collection of classes that form the building blocks of individual agents (Nwana et al., 1999). Agents are composed of five layers: API layer, definition layer, organisational layer, coordination layer and communication layer. The API layer allows interaction with the non-agentized world. The definition layer manages the task the agent must perform. The organisational layer manages the knowledge concerning other agents. The coordination layer manages coordination and negotiation with other agents. Finally, the communication layer allows the communication with other agents.

4. JADE

JADE (Java Agent DEvelopment framework) is a software framework to aid the development of agent applications in compliance with the FIPA specifications for interoperable intelligent MAS (Bellifemine et al., 2001). JADE is an open source project, and the complete system can be downloaded from JADE site <http://jade.cse.ist.it>.

The JADE system can be described from two different points of view. On the one hand, JADE is a runtime system for FIPA-compliant MAS, supporting application agents whenever they need to exploit some feature covered by the FIPA standard specification (message passing, agent life-cycle management, etc.). On the other hand, JADE is a Java framework for developing FIPA-compliant agent applications, making FIPA standard assets available to the programmer through object oriented abstractions. The two following subsections will present JADE from the two standpoints, trying to highlight the major design choices followed by the JADE development team.

4.1 Runtime System

JADE communication architecture tries to offer flexible and efficient messaging, transparently choosing the best transport available and leveraging state-of-the-art distributed object technology embedded within the Java runtime environment. While appearing as a single entity to the outside world, a JADE agent platform is itself a distributed system, since it can be split over several hosts with one of them acting as a front end where AMS and DF agents are placed. A JADE system comprises one or more Agent Containers, each living in a separate Java Virtual Machine (JVM) and delivering runtime environment support to some JADE agents (see Figure 13.2).

The JADE Runtime System tries to provide efficient and flexible messaging services to user applications. JADE distinguishes between inter-platform messaging (the sender and the receiver agents live on different platforms) and intra-platform messaging (the two interacting agents are within the same plat-

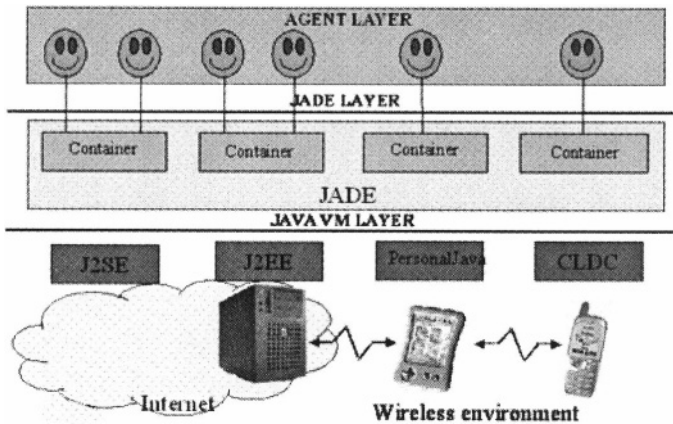


Figure 13.2. JADE agent platform

form). While inter-platform messaging has to comply with FIPA specifications, intra-platform message delivery is strictly a JADE issue, therefore JADE provides a Java interfaces that allows plugging the proper intra-platform transport protocol. Available implementations of this interface include Java RMI for intra-platform communications between J2EE and J2SE containers (i.e., when the container runs inside a J2EE or J2SE JVM) and a proprietary protocol when one of the container in running on a Personal Java or CLDC JVM (i.e., is running on a small device). In this last case, the container is split into two parts: one (front-end) running on the Personal Java or CLDC JVM and the other (back-end) running on a special container, called mediator, living on a J2EE or J2SE JVM (see Figure 13.3).

Since JADE is a distributed agent platform, its Agent Communication Channel (ACC) is split into different components, running on the different agent containers that make up the platform. The major features of JADE ACC are:

- Multiple MTPs can be deployed as plug-ins on multiple containers;
- One hop message routing for outgoing and incoming messages; and
- Protocol independent address caching.

The general JADE messaging framework allows to deploy new transport ports during normal platform operation: the JADE administrator can add a new protocol to any agent container, simply logging in the management GUI and providing the Java class that implements the MTP.

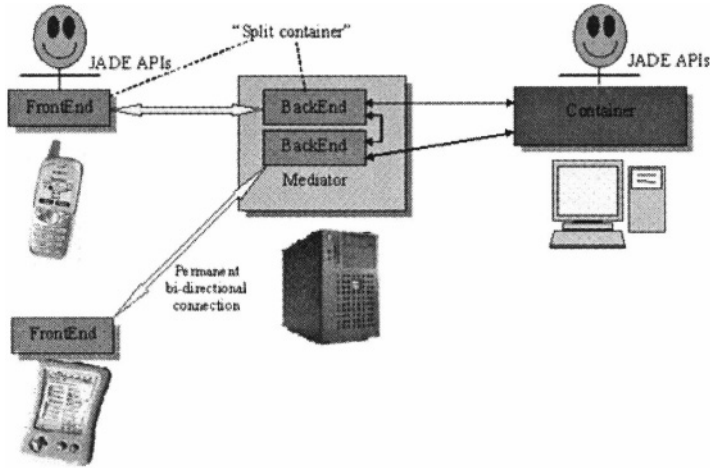


Figure 13.3. JADE communication architecture for small devices

An agent platform can now have any number of addresses, scattered around different hosts. Message routing support is needed to manage this rather general topology; the ACC provides a routing service that is guaranteed to require at most one hop. When a message reaches the platform through one of the available external communication ports, the ACC looks up the receiver agent ID to retrieve the agent container where it must dispatch the incoming ACL message. If the agent lives within the same container, the ACC uses an optimized local call, otherwise it relies on Java RMI.

When an agent wants to send a message to another, living on a different platform, it asks its local ACC for delivery service. The ACC reads the address list of the agent ID of the message recipient and tries all the addresses until one of them succeeds; for a specific address, the ACC discovers which MTP has to be used (FIPA addresses are URLs, so they contain a part that identifies the protocol) and checks to see whether that MTP is installed on the current agent container. If so, the locally available MTP is used, otherwise the ACC routes the message to a suitable container using a table that stores the deployment location of each MTP in the agent platform.

The JADE messaging subsystem also has an address caching feature that allows direct communication between agents without unnecessary table lookups: intra-platform addresses and standard FIPA addresses are cached on each container exactly in the same way. On cache hits, the messaging subsystem does not even need to know whether the receiver is local, intra-platform or inter-platform. The cache is updated according to an optimistic attitude (i.e., if a cached address becomes stale the message delivery operation fails with an ex-

ception and the cached item is refreshed) and the cache replacement policy is the usual Least Recently Used one.

JADE ACC can also be deployed on its own, without a complete agent container. This is meant to enable users to build and deploy agent level gateways and firewalls: a standalone ACC lives within a JVM that can route and filter ACL messages but cannot host FIPA agents.

4.2 Agent Model

FIPA specifications state nothing about agent internals, but when JADE was designed and built they had to be addressed. A major design issue is the execution model for an agent platform, both affecting performance and imposing specific programming styles on agent developers. As will be shown in the following, JADE solution stems from the balancing of forces from ordinary software engineering guidelines and theoretical agent properties.

A distinguishing property of a software agent is its autonomy; an agent is not limited to react to external stimuli, but it's also able to start new communicative acts of its own. A software agent, besides being autonomous, is said to be social, because it can interact with other agents in order to pursue its goals or can even develop an overall strategy together with its peers.

FIPA standard bases its ACL on speech-act theory (Searle, 1969) and uses a mentalistic model to build a formal semantic for the performatives agents exchange. This approach is quite different from the one followed by distributed objects and rooted in *design by contract* (Meyer, 1997); a fundamental difference is that invocations can either succeed or fail but a request speech act can be refused if the receiver is unwilling to perform the requested action.

Trying to map the aforementioned agent properties into design decisions, we can say:

- Agents are autonomous then are active objects;
- Agents are social then intra-agent concurrency is needed;
- Agent messages are speech acts then asynchronous messaging must be used; and
- Agents can say “no” then peer-to-peer communication model is needed.

The autonomy property requires each agent to be an active object (Lavender and Schmidt, 1996) with at least a Java thread, to proactively start new conversations, make plans and pursue goals. The need for sociality has the outcome of allowing an agent to engage in many conversations simultaneously, dealing with a significant amount of concurrency.

The third requirement suggests asynchronous message passing as a way to exchange information between two independent agents, that also has the ben-

enefit of producing more reusable interactions (Singh, 1999b). Similarly, the last requirement stresses that in a MAS the sender and the receiver are equals (as opposed to client/server systems where the receiver is supposed to obey the sender). An autonomous agent should also be allowed to ignore a received message as long as he wishes; this advocates using a pull consumer-messaging model (OMG, 2000a), where incoming messages are buffered until their receiver decides to read them.

The above considerations help in deciding how many threads of control are needed in an agent implementation; the autonomy requirement forces each agent to have at least a thread, and the sociality requirement pushes towards many threads per agent. Unfortunately, current operating systems limit the maximum number of threads that can be run effectively on a system. JADE execution model tries to limit the number of threads and has its roots in actor languages.

The Behaviour abstraction models agent tasks: a collection of behaviours are scheduled and executed to carry on agent duties. Behaviours represent logical threads of a software agent implementation. According to Active Object design pattern (Lavender and Schmidt, 1996), every JADE agent runs in its own Java thread, satisfying autonomy property; instead, to limit the threads required to run an agent platform, all agent behaviours are executed cooperatively within a single Java thread. So, JADE uses a thread-per-agent execution model with cooperative intra-agent scheduling.

JADE agents schedule their behaviour with a “cooperative scheduling on top of the stack,” in which all behaviours are run from a single stack frame (on top of the stack) and a behaviour runs until it returns from its main function and cannot be preempted by other behaviours (cooperative scheduling).

JADE model is an effort to provide fine-grained parallelism on everyday-hardware. A similar, stack based execution model is followed by Illinois Concert runtime system for parallel object oriented languages (Karamcheti et al., 1996). Concert executes concurrent method calls optimistically on the stack, reverting to real thread spawning only when the method is about to block, saving the context for the current call only when forced to.

Choosing not to save behaviour execution context means that agent behaviours start from the beginning every time they are scheduled for execution. So, behaviour state that must be retained across multiple executions must be stored into behaviour instance variables. A general rule for transforming an ordinary Java method into a JADE behaviour is:

- Turn the method body into an object whose class inherits from Behaviour;
- Turn method local variables into behaviour instance variables; and
- Add the behaviour object to agent behaviour list during agent startup.

The above guidelines apply the reification technique (Johnson and Zweig, 1991) to agent methods, according to Command design pattern (Lea, 1997); an agent behaviour object reifies both a method and a separate thread executing it. A new class must be written and instantiated for every agent behaviour, and this can lead to programs harder to understand and maintain. JADE application programmers can compensate for this shortcoming using Java anonymous inner classes; this language feature makes the code necessary for defining an agent behaviour only slightly higher than for writing a single Java method.

JADE thread-per-agent model can deal alone with the most common situations involving only agents: this is because every JADE agent owns a single message queue from which ACL messages are retrieved. Having multiple threads but a single mailbox would bring no benefit in message dispatching. On the other hand, when writing agent wrappers for non-agent software, there can be many interesting events from the environment beyond ACL message arrivals. Therefore, application developers are free to choose whatever concurrency model they feel is needed for their particular wrapper agent; ordinary Java threading is still possible from within an agent behaviour. The developer implementing an agent must extend Agent class and implement agent-specific tasks by writing one or more Behaviour subclasses. User defined agents inherit from their superclass the capability of registering and deregistering with their platform and a basic set of methods (e.g., send and receive ACL messages, use standard interaction protocols, register with several domains). Moreover, user agents inherit from their Agent superclass some methods to manage the agent behaviours.

JADE contains ready made behaviours for the most common tasks in agent programming, such as sending and receiving messages and structuring complex tasks as aggregations of simpler ones. JADE recursive aggregation of behaviour objects resembles the technique used for graphical user interfaces, where every interface widget can be a leaf of a tree whose intermediate nodes are special container widgets, with rendering and children management features. An important distinction, however, exists: JADE behaviours reify execution tasks, so task scheduling and suspension are to be considered, too.

Thinking in terms of software patterns, if Composite is the main structural pattern used for JADE behaviours, on the behavioural side we have Chain of Responsibility: agent scheduling directly affects only top-level nodes of the behaviour tree, but every composite behaviour is responsible for its children scheduling within its time frame.

4.3 Management and Testing Tools

In addition to a runtime library and an agent programming library, JADE offers some tools to manage the running agent platform and to monitor and

debug agent societies. All these tools are implemented as agents themselves, and they require no special support to perform their tasks, they simply rely on JADE AMS. The general management console for a JADE agent platform is called RMA (Remote Management Agent). The RMA acquires the information about the platform and executes the GUI commands to modify the status of the platform (creating new agents, shutting down peripheral containers, etc.) through the AMS. On the one hand, the RMA asks the AMS to be notified about the changes of state of platform agents, on the other hand, it transmits to the AMS the requests for creation, deletion, suspension and restart received by the user. The Directory Facilitator agent also has a GUI of its own, with which the DF can be administered, adding or removing agents and configuring their advertised services.

The graphical tools with which JADE users can debug their agents are the Dummy Agent, the Sniffer Agent, and the Introspector Agent.

The Dummy Agent is a simple, yet very useful, tool for inspecting message exchanges among agents. The Dummy Agent facilitates validation of an agent message exchange pattern before its integration into a MAS and facilitates interactive testing of an agent. The graphic interface provides support to edit, compose and send ACL messages to agents, to receive and view messages from agents, and, eventually, to save/load messages to/from disk.

The Sniffer Agent makes it possible to track messages exchanged in a JADE agent platform. When the user decides to sniff a single agent or a group of agents, every message directed to or coming from that agent or group is tracked and displayed in the sniffer window, using a notation similar to UML Sequence Diagrams. Every ACL message can be examined by the user, who can also save and load every message track for later analysis.

The Introspector Agent, finally, is a very powerful tool that allows to debug and introspect a running agent through the following functionalities: (i) monitor and control the agent life-cycle; (ii) inspect all its exchanged messages, both the queue of sent and received messages; and (iii) monitor the queue of behaviours, including the possibility of executing a behaviour step-by-step, in a similar way to a code debugger.

4.4 Applications

JADE is being used in a plethora of projects and applications, both from the academic and the industrial communities. In particular, it has been used and is used in four projects sponsored by the European Commission: FACTS, CoMMA, LiMe and Agencities.RTD.

FACTS is a project in the framework of the ACTS programme of the European Commission that used JADE in two application domains. In the first application domain, JADE provides the basis for a new generation TV enter-

tainment system. The user accesses a MAS to help him on the basis of his profile that is captured and refined over time through the collaboration of agents with different capabilities. The second application domain deals with agents in collaboration, and at the same time competing, in order to help the user to purchase a business trip. A Personal Travel Assistant represents the interests of the user and cooperates with a Travel Broker Agent in order to select and recommend the business trip.

LiMe is a Long Term Research Programme of the European Commission under its “I-cubed” (Intelligent Information Interfaces) programme LiMe. The goal of the project was to create a MAS for the enhancement of social interaction within connected communities. JADE has successfully supported the LiMe communicating agents for dynamic user profiling, collective information dissemination and memory management for a 2-day field trial.

CoMMA is a going project in the framework of the IST programme of the European Commission that is using JADE to help users in the management of an organization corporate memory and in particular to facilitate the creation, dissemination, transmission and reuse of knowledge in the organization. The main objective of this project is to implement and test a Corporate Memory Management framework based on agent technology. The innovative aspect of the project is to integrate several emerging technologies that were generally used separately until now: agent technology, knowledge modeling, XML technology, information retrieval techniques and machine learning techniques. Integration of these technologies in one system is already a challenge yet another challenge is the definition of the methodology supporting the whole design process. The project intends to implement the system in the context of two scenarios (*i*) the insertion of new employees in the company; and (*ii*) the technology monitoring.

Agentcities.RTD is a going project in the framework of the IST programme of the European Commission. The project's objectives are to create an on-line, distributed testbed to explore and validate the potential of agent technology for future dynamic service environments. The project aims to produce the following important results: (*i*) an open, stable, scalable and reliable network architecture that allows standards compliant agents to discover each other, communicate and offer services to one another; (*ii*) models, methodology and prototype solutions for the integration of business services into the service environment; and (*iii*) practical methodologies for the application of agent communication technologies (semantic models, ontology, expression of content and protocols) to service modeling in open heterogeneous environment. Currently, the Agentcities network counts 160 registered platforms. The platforms are based on more than a dozen of heterogeneous technologies, including Zeus, FIPA-OS, Comtec (see <http://ias.comtec.co.jp/ap>), AAP (see <http://sf.us.agentcities.net/aap>), Opal. More than 2/3 of them are

based on JADE and its derived technologies, as LEAP (Adorni et al., 2001) and BlueJADE (see <https://sourceforge.net/projects/bluejade>).

Rockwell Automation uses JADE for manufacturing control and automation. JADE is the middleware for their tool MAST – Manufacturing Agent Simulation Tool (Marik et al., 2003) which allows to model manufacturing systems as an holonic system with respect to the material-handling tasks, i.e., the transportation of entities among different manufacturing cells using different means of transport, mainly the conveyor belts and AGVs (Automated Guided Vehicles).

5. Conclusions

In this chapter, we introduced FIPA-compliant development tools and, in particular, we presented JADE (Java Agent DEvelopment framework), a software environment to build agent systems for the management of networked information resources in compliance with the last FIPA specifications.

FIPA is an international non-profit association of companies and organizations sharing the effort to produce specifications for generic agent technologies. FIPA does not promote a technology for just a single application domain but a set of general technologies for different application areas that developers can integrate to make complex systems with a high degree of interoperability. FIPA started its activity in the 1997 and defined a set of specifications that are now considered a “de facto” standard for agent interoperability.

For this reason, on the one hand, people working on agent development tools and libraries consider to be “FIPA-compliant” a key feature of their products and, on the other hand, people working on the realization of agent systems use more and more FIPA-compliant development tools and software libraries.

In particular, JADE seems to be one of the most appreciated and used FIPA-compliant development tool. It may be proved, by the number of users of any part of the world that are using JADE today.

Acknowledgements

Thanks to all those people who have and continue to contribute to the development of JADE. TILAB has been partially supported for this work by the Italian M.I.U.R. through the Te.S.C.He.T. Project (Technology System for Cultural Heritage in Tourism). University of Parma has been supported for this work by TILAB and by the European Commission through the contract IST-2000-28385 – Agentcities.RTD.

Chapter 14

COORDINATION INFRASTRUCTURES IN THE ENGINEERING OF MULTIAGENT SYSTEMS

Andrea Omicini, Sascha Ossowski and Alessandro Ricci

Abstract On the theoretical side, coordination is a critical issue for MAS engineering, since it deals with modelling and managing the ever growing complexity of the agent interplay within a MAS. On the practical side, the availability of powerful and robust infrastructures is a key factor to enable and promote MAS as a mainstream software engineering technology. By adopting Activity Theory as a unifying framework for the many existing approaches to MAS coordination, we put forward the notion of *artifact* as a key concept for infrastructures, from which we derive some distinctive properties that a coordination infrastructure should feature. Finally, we discuss how a principled approach to MAS engineering based on coordination infrastructures could be built around such a notion.

1. Introduction

Coordination is one of the key issues in the modelling and engineering of complex systems, and has been the subject of numerous investigations in areas such as Sociology, Economics and Organisational Theory. From an engineering point of view, the question of how to *design* computational mechanisms that allow for efficient coordination is foremost: coordination is conceived as a means to integrate various activities or processes in such a way that the resulting ensemble shows desired characteristics and functionalities. The design of coordination mechanisms is particularly challenging in the field of MAS, as they are usually embedded in highly dynamic environments, and neither the number nor the behaviour of agents can be directly controlled at design time.

In this chapter, we discuss how coordination infrastructures can be used as a means to instill coordination in open multiagent systems. In particular, we claim that additional high-level abstractions need to be integrated into agent-oriented design methodologies in order to exploit the full potential of coordi-

nation infrastructures, and to engineer coordinated MAS in open environments in an efficient and principled manner.

The chapter is organised as follows. Section 2 argues that the key problem of coordination engineering in MAS amounts to the governance of interaction from both the agents' and the designer's point of view. Subsequently (section 3) we outline the role of coordination infrastructures for this task and point to shortcomings in current approaches. Setting out from findings in Activity Theory, section 4 provides a uniform conceptual framework for many different approaches to coordination, and introduces the notion of *coordination artifact* as a key abstraction, that allows for a smooth conceptual integration of coordination infrastructures into MAS design. Section 5 provides clues on how to engineer MAS based on advanced coordination infrastructures within such an integrated conceptual framework. Final discussion concludes the chapter.

2. Coordination in MAS

2.1 Models of Coordination in MAS

Maybe the most widely accepted conceptualisation of coordination in the MAS field originates from work in the area of Organisational Science. (Malone and Crowston, 1994), define coordination as *the management of dependencies* between organisational activities. One of the many workflows in an organisation, for instance, may involve a secretary writing a letter, an official signing it, and another employee sending it to its final destination. The interrelation among these activities is modelled as a *producer/consumer* dependency, which can be managed by inserting additional *notification* and *transportation* actions into the workflow.

It is straightforward to generalise this approach to coordination problems in multiagent systems. Obviously, the subjects whose activities need to be coordinated (sometimes called *coordinables*) are the agents. The entities between which dependencies arise (or *objects of coordination*) are often termed quite differently, but usually come down to entities like goals, actions and plans. Depending on the characteristics of the MAS environment, a taxonomy of dependencies can be established, and a set of potential coordination actions assigned to each of them, e.g., (von Martial, 1992). Within this model, the *process* of coordination is to accomplish two major tasks: first, a *detection* of dependencies needs to be performed, and second, a *decision* respecting which coordination action to apply must be taken. A coordination *mechanism* shapes the way that agents perform these tasks (Ossowski, 1999).

The dependency model of coordination appears to be particularly well suited to *represent* relevant features of a coordination problem in MAS. The TAEMS framework presented by (Decker, 1996a), for instance, has been used to model coordination requirements in a variety of interesting MAS domains. It is also

useful to rationalise observed coordination behaviour along the lines of the knowledge-level perspective put forward by (Newell, 1993). Still, when *designing* coordination processes for real-world MAS, things are not as simple as the dependency model may suggest. Dependency detection may come to be a rather knowledge intensive task, which is further complicated by incomplete and potentially inconsistent local views of the agents. Moreover, making timely decisions that lead to efficient coordination actions is also everything but trivial. The problem becomes even more difficult when agents pursuing partially conflicting goals come into play. In all but the most simple MAS, the instrumentation of these tasks gives rise to complex patterns of interactions among agents. The set of possible interactions is often called the *interaction space* of coordination.

From a software engineering perspective, coordination is probably best conceived as the effort of *governing the space of interaction* of a MAS (Busi et al., 2001). When approaching coordination from a *design* stance, the basic challenge amounts to how to make agents converge on interaction patterns that adequately (i.e., instrumentally with respect to desired features of the agents and/or the MAS as a whole) solve the dependency detection and decision tasks.

2.2 Objective vs. Subjective Coordination in MAS

There are two ways of looking at the space of interaction: from the inside and from the outside of the interacting entities. In the context of multiagent systems, this amounts to say that we can look at interaction within a MAS from either the viewpoint of an agent, or from the viewpoint of an external observer not directly involved in the interaction. According to (Schumacher, 2001), and (Omicini and Ossowski, 2003), these are called, respectively, *subjective* and *objective* viewpoints over coordination.

From the subjective viewpoint of an agent, the space of interaction basically amounts to the observable behaviour of other agents and the evolution of the environment over time, filtered and interpreted according to the individual agent's perception and understanding. From the objective viewpoint, the space of agent interaction is roughly given by the observable behaviour of all the agents of a MAS and of the agent environment as well, and by their mutual interactions – more precisely, by all their *interaction histories* (Wegner, 1997). When adopting the acceptance of MAS coordination as the governance of the agent interaction space, then the two different viewpoints lead to two different ways of coordinating.

When looking at interaction from the individual viewpoint of an agent, *subjective coordination* roughly amounts to (i) monitoring all interactions that are perceivable and relevant to the agent, keeping track of their evolution over time; and (ii) finding out which (sequence of) actions would bring the over-

all state of the MAS (or, more generally, of the agent's world) to match the agent's own goals. So, in general, the acts of an agent that coordinates within a MAS are driven by its own perception and understanding of the behaviour of the other agents', capabilities and goals, as well as of the environment state and dynamics.

On the other hand, when taking an external viewpoint over interaction in a MAS – typically, the designer's viewpoint –, *objective coordination* means either directly or indirectly acting upon agent interaction so as to make the resulting evolution of a MAS accomplish one or more of the observer's (e.g., MAS designer's) goals. In general, the acts of external observers – whether they be MAS designers, developers, users, managers, or even agents working at the meta-level – are influenced not only by their perception and understanding of MAS agents and environment, but also by their a-priori knowledge of the agents' aims, capabilities and behaviour. Furthermore, some form of prediction of the global behaviour of the MAS and its environments is often desirable (Ossowski et al., 2002), so as to instill a coordination that is effective over time from the standpoint of the user.

2.3 Implications for MAS Engineering

Subjective and objective coordination have a different impact over MAS engineering. Subjective coordination affects the way in which individual agents behave and interact, whereas objective coordination affects the way in which interaction among the agent and the environment is enabled and ruled. So, whereas the main focus of subjective coordination is the behaviour of agents as (social) *individuals* immersed in a MAS, the emphasis of objective coordination lies more on the behaviour of a MAS as a whole.

When designing the architecture and the inner dynamics of single agents, the subjective viewpoint on coordination is clearly the most pertinent one. How to model other agents' mental states and to predict their actions, how to interpret and handle shared information in the agent system, when and why to move from an agent environment to another, and so on – all these questions concern subjective coordination, and affect the way in which the agents of a MAS are designed, developed and deployed as individual entities. So, the viability of approaches adopting a subjective coordination viewpoint to the engineering of MAS strictly depends not only on the mental (reasoning, planning and deliberation) capabilities of the agents, but also on their ability to foresee the effect of their actions on the environment, the behaviour of the other agents, and the overall dynamics of the environment as well.

On the other hand, in principle an external observer does not directly interact with the agents of a MAS. As a result, some capability to act on the space of MAS interaction without dealing directly with agents is obviously required

in order to enable any form of objective coordination. Given that agents are typically situated entities, acting on the agent environment makes it possible to affect the behaviour of an agent system without having to alter the agents themselves. Under this acceptance, then, objective coordination deals with the agent environment: modifying the virtual machine supporting agent functioning, changing resource availability and access policies, altering the behaviour of the agent communication channel, be it virtual or physical, and so on – all these are possible ways to influence and possibly harness the behaviour of a MAS without directly intervening on individual agents and undermine the basic assumption of agent autonomy. The viability of objective coordination in the engineering of agent systems depends then on the availability of suitable models of the agent environment, and on their proper embodiment within agent infrastructures. There, objective coordination would conceivably take on the form of a collection of suitably expressive coordination abstractions, provided as run-time coordination services by the agent infrastructure.

As discussed by (Omicini and Ossowski, 2003), the engineering of a MAS requires that both subjective and objective coordination are blended together. On the one side, in fact, a purely subjective approach to coordination in the engineering of agent systems would endorse a mere reductionistic view, coming to say that agent systems are compositional, and their behaviour is nothing more than the sum of the individual's behaviour – an easily defeasible argument, indeed. Among the many consequences, this would require global properties of the agent system to be “distributed” among individuals, providing neither abstractions nor mechanisms to encapsulate such properties. As a result, the purely subjective approach would directly entail lack of support for design, development, and, even more, deployment of agent systems' global properties – which would result in substantial difficulties for incremental development, impractical run-time modification, and so on. On the other side, a purely objective approach to coordination in the engineering of agent systems would endorse a rough holistic view – where only inter-agent dependencies and interactions count, and individuals' behaviour has no relevance for global system behaviour. Among the many consequences, this would stand in stark contrast with any notion of agent autonomy, and would prevent agents from featuring any ability to affect the environment for their own individual purposes – no space for anything resembling an agent left, in short.

In the end, all the above considerations suggest that any principled approach to the engineering of agent systems should necessarily provide support for both subjective and objective models of coordination, possibly integrating them in a coherent conceptual framework, and providing at the same time a suitable support for all the phases of the engineering processes – in terms of coordination languages, development tools, and run-time environments.

3. Infrastructures for MAS Engineering

3.1 On the Notion of Infrastructure

Today, infrastructure is a fundamental notion for complex systems in general, not only in computer science and engineering, but also in the context of organisational, political, economical and social sciences. In its most general acceptance, an infrastructure is defined as:

(*Merriam-Webster*) † (1) the underlying foundation or basic framework (as of a system or organisation) (2) the permanent installations required for military purposes; (3) the system of public works of a country, state, or region; also: the resources (as personnel, buildings, or equipment) required for an activity;

(*Cambridge*) † (4) the basic systems and services, such as transport and power supplies, that a country or organisation uses in order to work effectively;

(*The American Heritage*) † (5) the basic facilities, services, and installations needed for the functioning of a community or society, such as transportation and communications systems, water and power lines, and public institutions including schools, post offices, and prisons.

Every definition underlines the role of infrastructure as (part of) the environment that provides basic resources and critical services to complex systems (such as organisations, communities, societies, countries) living on top of it. In particular, definition (2) remarks the fact that an infrastructure is a *persistent* entity: once installed, an infrastructure typically survives the many systems it supports. Also, definitions (4) and (5) remark the key role of infrastructures: their services typically cover critical system issues, and provide features that individual system components could not afford to provide or obtain elsewhere.

In the context of MAS, infrastructure obviously plays a key role, given the potential complexity of both the system components (agents) and the component interplay (agent societies).

(Gasser, 2001), defines an infrastructure as

“a technical and social substrate that stabilises and rapidly enables instrumental (domain-centric, intentional) activity in a given domain... (solving) typical, costly, commonly accepted community (technical) problems in a systematic and appropriate ways ”

Here, it is important to emphasise the notion of infrastructure as a *social, enabling* support for providing MAS with cheap and systematic solutions to common problems.

Another interesting definition is provided by (Sycara et al., 2003):

“Agents in a MAS are expected to coordinate by exchanging services and information, to be able to follow complex negotiation protocols, to agree on commit-

ments and to perform other socially complex operations. We define the infrastructure of a MAS as the set of services, conventions, and knowledge that support such complex interactions."

The stress is here on the support of complex agent (social) interplay, which is expressed in terms of services, convention and knowledge.

3.2 The Role of MAS Infrastructure

In a more abstract acceptance than the ones above, the main role of infrastructures in MAS is to model and shape the *agent environment*, from the two points of view (i) of the agents living in the MAS; and (ii) of MAS designers. From the inner viewpoint of an individual agent, the infrastructure typically provides the means to deal with the agent environment: to perceive and affect its state and dynamics (in general), to access resources and services, to obtain and store information, to interact with other agents (in particular). Typically, a suitably expressive and well-engineered infrastructure allows agents to represent their environment only through the runtime abstractions provided by the infrastructure, and to modify the agent environment according to the agent's needs and goals through infrastructure services. From the external viewpoint of a human designer, MAS are typically open systems, both in terms of the unpredictability of their environment (due to components and interactions not under the control of MAS designers), and of the dynamism of both MAS structures (e.g., the set of agents in a MAS) and MAS processes as well (e.g., the coordination activities within a MAS). Infrastructures are then the suitable place for designers to embed elements of control of MAS despite their inherent openness: such control can be exerted by means of runtime abstractions provided by the infrastructure that can embody and enforce interaction constraints, coordination laws and social norms. Even more, once they are suitably described and made accessible to agents, the same runtime abstractions can be exploited by intelligent agents in order to represent coercive structures of a MAS, and to act upon its global behaviour by introducing and/or modifying constraints, laws and norms (Omicini and Ricci, 2003).

Infrastructures play then a key role in the engineering of MAS, too. This is quite obvious when considering the last stages of the engineering process, that is, the development and deployment of MAS. Nevertheless this also holds when taking the early stages into account, that is, the modelling and design of MAS: the abstractions provided by the infrastructure are the most natural candidates to be adopted and exploited in the design of MAS structures and activities, which are then to be engineered on top of such abstractions. So, runtime abstractions should be flexible enough to support the engineering of heterogeneous systems, and – at the same time – effective in minimising the gap between the design and development / deployment / runtime of systems.

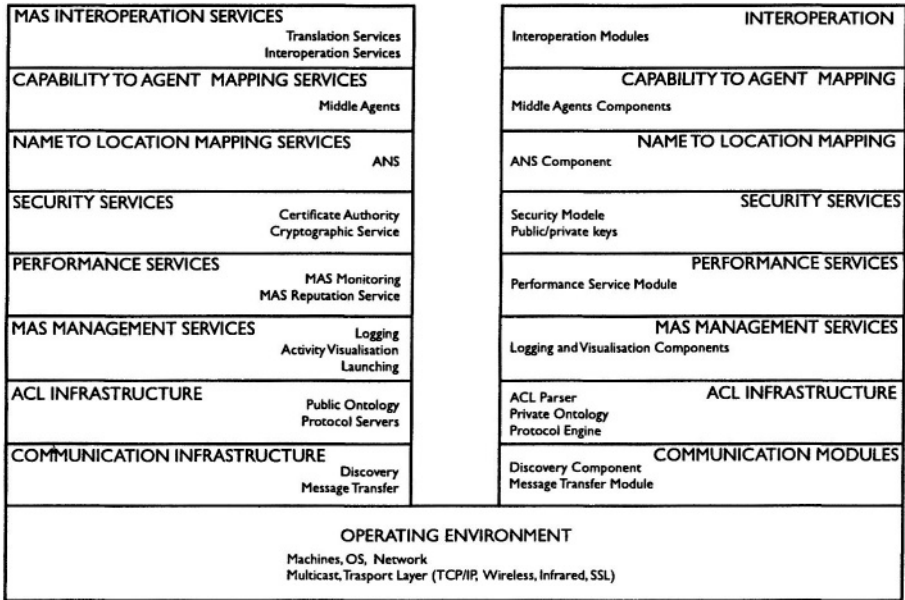


Figure 14.1. MAS infrastructure levels, according to (Sycara et al., 2003)

In this context, the tools provided by an infrastructure are fundamental to enable the manipulation of the abstractions through all the engineering stages, in particular at runtime. The definition of the engineering tools is a primary issue, that should be necessarily inspired and driven by the model embodied by the MAS infrastructure itself (Denti et al., 2002).

In the end, MAS infrastructures and tools play an essential engineering role by *keeping abstractions alive* through the whole engineering process, thus enabling software engineers to first design and then observe and act on MAS structures and processes at runtime, working upon abstractions adopted and exploited for the design of a MAS. This feature is particularly important to support forms of *online engineering* (see chapter 18), i.e., the capability of supporting system design / development / evolution while the systems are running – a particularly relevant feature in the context of MAS, given their intrinsic complexity and openness.

3.3 Enabling vs. Governing Infrastructures

As discussed above, infrastructures are useful to encapsulate and support critical features and properties of MAS; these properties typically concern the *interaction* dimension. For this extent, current MAS infrastructures can be

considered *enabling infrastructure*, since they provide abstractions that basically enable agent interaction at different levels: from communication to interoperability, to basic interaction services. This is apparent when considering the abstract architecture of two of the most important infrastructures currently adopted for MAS development and deployment: RETSINA (Sycara et al., 2003) (bottom of Figure 14.2) and JADE (Bellifemine et al., 2001) (top of Figure 14.2). There, in fact, services like agent communication, inter-operation, security, naming, location, etc., are necessary preconditions that make it possible for agents to live, coexist and interact within a MAS. Enabling infrastructures, then, basically define the nature of the agent interaction space within a MAS.

However, the increasing complexity and articulation of MAS for today's application scenarios call for a most effective engineering support from infrastructure, beyond the mere enabling of agent interaction. A well known example are Electronic Institutions (Noriega and Sierra, 2002): the social and normative capabilities required to infrastructures supporting eInstitutions goes far beyond the services provided by general purpose MAS enabling infrastructures, and cannot be straightforwardly engineered on top of it. Another example comes from team-oriented coordination: in order to be independent from the specific agent model, the TEAMCORE approach introduces the PROXY abstraction, an infrastructure component provided to agents for managing automatically all coordination dependencies with respect to the teams that agents belong to (Tambe et al., 2000). Similar team-oriented capability has been added to RETSINA by enhancing its Individual Agent Architecture (Giampapa and Sycara, 2002): in this way, contrary to the TEAMCORE approach, no real infrastructure support is provided from the infrastructure to team-oriented coordination, since the team-oriented capability is obtained by relying on augmented capabilities of the individual agents.

In the end, current general purpose MAS infrastructures typically lack suitably abstractions to *govern* agent interaction. This seems instead a fundamental feature for enabling the specification and enactment of social norms, but also – more generally – for defining and executing social activities, such as agent coordination. In other words, complex system engineering calls for *governing infrastructures*, providing flexible and robust abstractions to model and shape the agent interaction space, in accordance with the social and normative objectives of systems.

Governing infrastructures become the natural *loci* where to embody a conceptual framework that uniformly accounts for organisation, coordination and security of MAS altogether (Omicini et al., 2003). From the organisational point of view, infrastructures are to provide explicit abstractions for modelling the structure of an organisation and its rules – e.g., using the notion of role and related permissions to access to resources. This is the case, for instance, of

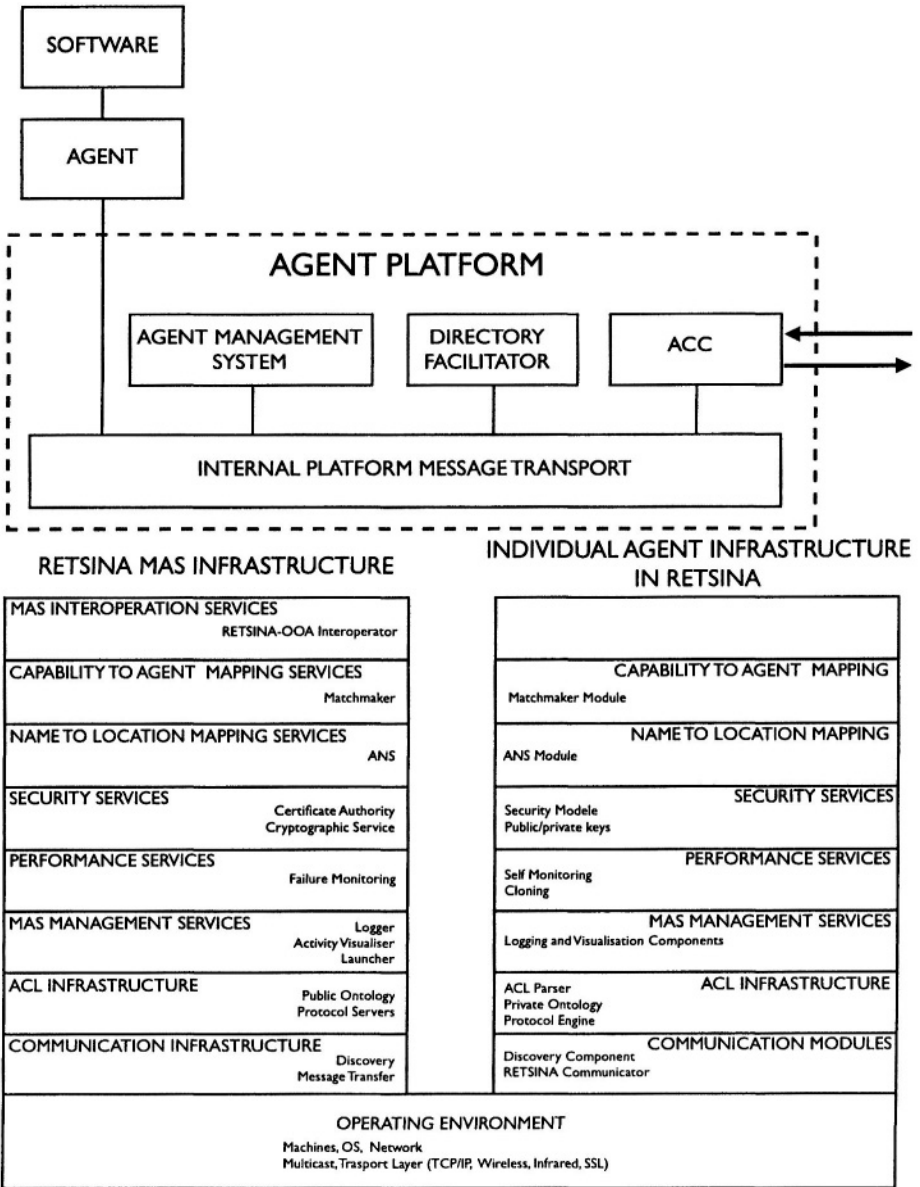


Figure 14.2. (top) FIPA reference model for Agent Platform, adopted by JADE (see chapter 13) – where the ACC is the Agent Communication Channel Component – and (bottom) RETSINA functional levels (Sycara et al., 2003)

the information system infrastructure that support the RBAC model (Sandhu et al., 1996), which is attracting attention also in the context of MAS. From the coordination point of view, infrastructure support can be described effectively by adopting the notion of *coordination as a service* (Omicini and Ossowski, 2003; Viroli and Omicini, 2003): according to this vision, the infrastructure itself is the provider of runtime (coordination) abstractions designed for specifically supporting the specification, execution and maintenance of MAS social activities. These abstractions become a fundamental tool to face the engineering complexity of coordination in MAS: both from the designer's and the agents' point of view, the coordination burden is distributed between agents and the specialised services provided by the infrastructure. The expressiveness and flexibility of coordination abstractions strongly influence the engineering of social activities, and, consequently, the complexity of the solutions adopted for the challenging application scenarios. Since they are part of the infrastructure, these coordination abstractions are typically expected to be robust and reliable, and specifically designed to support a critical activity as coordination is.

Two observations are worthwhile here. Firstly, the evolution from enabling to governing infrastructures can be devised also in other computer science fields, characterised as well by complex organisations and collaboration activities: CSCW and Workflow Management are relevant examples. Especially in the CSCW context the need for suitable infrastructure support for coordination has already emerged as a fundamental issue. (Schmidt and Simone, 1996), for instance, identify basic properties that coordination abstractions provided by an infrastructure should feature. Secondly, the approach of coordination as a service has also a deep impact on AOSE methodologies, since coordination abstractions – as they embody the *social* aspect of MAS – are meant to become explicitly subject of all the engineering stages, as it happens in SODA methodology (Omicini, 2001).

4. Modelling Coordination Infrastructures with Activity Theory

The research on coordination infrastructures is a primary issue also in other disciplines focusing on complex collaborative works in articulated organisation, such as CSCW and organisational science. The models and theories adopted and developed in those contexts can provide then useful insight for the MAS context. Accordingly, we considered Activity Theory very effective to frame and analyse coordination activities inside an organisation context, and the infrastructure support they require.

4.1 Activity Theory as a Framework for MAS Coordination

Once the many different coordination approaches have been properly understood and classified, a uniform conceptual framework is required that suitably reconciles both objective and subjective coordination, and helps putting them in the best perspective in the context of MAS engineering. To this end, (Ricci et al., 2003) adopt *Activity Theory* in order to shed some light on the role of subjective and objective approaches to coordination engineering, and their mutual relationship.

Activity Theory (AT henceforth) is a social psychological theory about the developmental transformation and dynamics in collective human work activity (Leontjev, 1978; Vygotskij, 1978). AT focuses on *human activities*, which are distinguished by their respective (physical and ideal) *objects*, that give them their specific directions, i.e., the *objectives* of the activities. Cooperation is understood as a *collaborative activity*, with one objective, but distributed onto several actors, each performing *actions* accordingly to the shared objective. Explicit norms and rules regulate the relationships among the individual participants' work.

Central to AT is the notion of *artifact* as a mediator for any sort of interaction in human activities: artifacts can be either physical or cognitive, such as operating procedures, heuristics, scripts, individual and collective experiences, and languages. Artifacts embody a set of social practise: their design reflects a history of particular use. As mediating tools, they have both an *enabling* and a *constraining* function: on the one hand, artifacts expand out possibilities to manipulate and transform different objects, but on the other hand the object is perceived and manipulated not 'as such' but within the limitations set by the tool. (Ricci et al., 2003) define the notion of *coordination artifact* to identify artifacts that are used in the context of collaborative activities in particular, mediating the interaction among actors involved in the same social context. Coordination artifacts can be *embodied* or *disembodied*, referring to respectively physically or cognitive/psychological artifacts. A similar concept can be found also in the CSCW context, with the notion of coordinative artifacts (Schmidt and Simone, 2000). It is worth noting the different acceptance of the term artifact as used in AT and CSCW with respect to the traditional software engineering context (Barthelmeß and Anderson, 2002): in the latter, the term artifact is typically used to refer to documents (or deliverables) that are produced throughout a process, and the term *tool* is used to identify the means to perform operation on artifacts.

As far as collaborative activities are concerned, AT identifies three hierarchical levels defining their structure: *co-ordinated*, *co-operative*, and *co-constructive* (Bardram, 1998; Engeström et al., 1997).

- The *co-ordinated* aspect of work captures the normal and routine flow of interaction. Participants follow their *scripted roles*, each focusing on the successful performance of their actions, implicitly or explicitly assigned to them; they share and act upon a common object, but their individual actions are only externally related to each other. *Scripts* coordinating participants' actions are not questioned or discussed, neither known and understood in all their complexity: in this stage actors act as “wheels in the organisational machinery” (Kuutti, 1991), and co-ordination ensures that an activity is working in harmony with surrounding activities.
- The *co-operative* aspect of work concerns the mode of interactions in which actors focus on a common object and thus share the objective of the activity; unlike the previous case, actors do not have actions or roles explicitly assigned to them: with regard to the common object, each actor has to balance his/her own actions with other agent actions, possibly influencing them to achieve the common task. So, in this case the object of the activity is stable and agreed upon: however the means for realising the activity is not yet defined.
- The *co-constructive* aspect of work concerns interactions in which actors focus on re-conceptualising their own organisation and interaction in relation to their shared objects. Neither the object of work, nor the scripts are stable, and must be collectively constructed, i.e., co-constructed.

It is worth here to notice that in the analysis of collaborative activities, AT emphasises that a collaborative activity is not to be seen in general at one single level: co-ordination, co-operation, and co-construction are instead to be interpreted as *analytical* distinctions of the same collaborative activity, concurring in different times and modes to its development.

In the context of MAS coordination, the three levels identified by AT can be re-interpreted as follows (Figure 14.3):

- **Co-construction** – agents understand and reason about the (social) objectives (goals) of the MAS, and define a model of the social tasks required to reach them. This implies also identifying the interdependencies and the interactions to be faced and managed;
- **Co-operation** – agents design and define the coordination artifacts – either embodied (coordination media) or disembodied (plans, interaction protocols, etc.) – useful to carry on the social tasks and to manage the interdependencies and interactions devised out at the previous (co-construction) stage; and

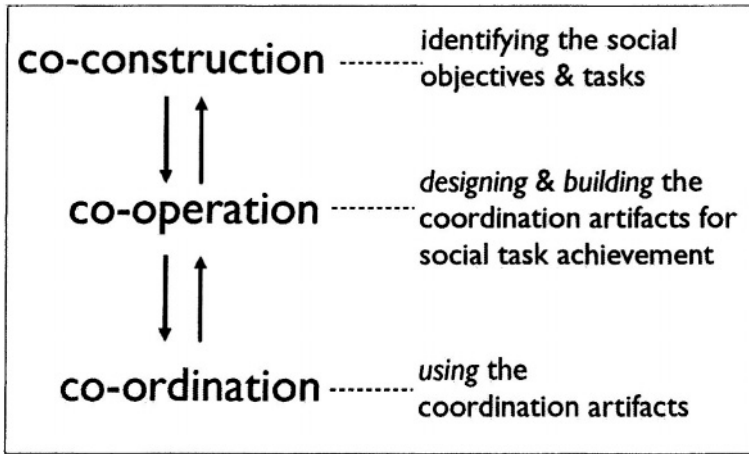


Figure 14.3. Levels of a collaborative activities as identified by Activity Theory and their relationship with coordination artifacts

- **Co-ordination** – agents exploit the coordination artifacts, and then the activities to manage interdependencies and interactions, either designed a-priori or planned at the co-operation stage, are enforced/automated.

At every level both top-down and bottom-up approaches are present: the former in modelling/designing/enacting social tasks, and the latter in identifying and managing dependencies and interactions. Both approaches rely on the engineering of coordination artifacts, be it embodied or disembodied.

Subjective approaches are fundamental for the co-construction and, in particular, the co-operation stage. Here it is necessary to reason about what kind of coordination is required, what kind of coordination laws must be developed to manage interactions and fulfil the social tasks identified in co-construction stage. Agent intelligence is useful to cooperatively build – by means of negotiation and high level (semantics driven) interaction protocols – effective coordination artifacts to be used in the co-ordination stage, be they disembodied such as interaction protocols or embodied such as coordination media. Instead, objective coordination is fundamental for the co-ordination stage, where coordination must be enacted in the most automated, fluid, and possibly optimised manner. The coordination medium abstraction (and coordination laws defining its behaviour) represents effectively the concept of embodied coordination artifact (and related mediating tools), embedding and enacting in the co-ordination stage the social laws and interaction constraints established in the co-operation stage.

Drawing a parallel between AT artifacts and coordination media may help to better recognise the role of the media inside MAS: as the artifacts, coordi-

nation media first are used to *enable* the interactions among the agents, and then to *mediate* them in the most automated manner. As the artifacts, media become *the* place where the *coordination knowledge* of the MAS is explicitly represented (Ossowski and Omicini, 2002), where it is enacted and can be further inspected. So, media become the source of the “social intelligence” that actually characterises the systemic/synergistic (as opposed to compositional) vision of MAS (Ciancarini et al., 2000). In this context, coordination laws become the coercive structures that can be used to tune and adapt dynamically such a collective intelligence (Ossowski, 1999).

4.2 Artifacts and Coordination Infrastructures

Quite frequently in the context of MAS, agents are the *only* abstraction used for system engineering – especially at the development and deployment stage. The matchmaking and brokering services required by any open MAS, for instance, are usually provided by *middle-agents* (Klusck and Sycara, 2001). Accordingly, these agents constitute a suitable way to embody AT artifacts at the co-ordination stage. So, in principle, they may take over the role of coordination media in the mediation of agent interactions.

However, AT clearly distinguishes between ontological properties of the artifacts (as well as related mediating tools) and the actors designing/developing (co-operation stage) and exploiting (co-ordination stage) the artifacts. This suggests to draw a similar distinction between agents and coordination media. As opposed to agents, the main properties that a coordination medium is expected to exhibit are the following:

- *Inspectability* – the behaviour of a coordination medium should be inspectable, both for human and artificial agents. Moreover, coordination specifications should be described in a declarative way, possibly with a formally defined semantics, to allow for their interpretation at run-time.
- *Efficiency/specificity* – a coordination medium should be specialised in the management of interactions, in order to maximise performance in the application of the coordination rules. Moreover, a medium should be specialised to support the concurrent actions (communications) of multiple agents, possibly providing security, reliability and fault tolerance capabilities.
- *Predictability* – the behaviour of a coordination medium should exactly reflect the coordination laws upon which it has been forged (autonomous, unpredictable behaviour is typically *not* desired). A formal semantics should be defined for the coordination model to precisely define the effect of the coordination laws on the state of the medium and, more generally, on the agent interaction space.

- *Malleability* – a coordination medium should be malleable, i.e., it should allow its behaviour to be forged and changed dynamically at execution time, according to the need. This property is fundamental for facing the openness of MAS environment, in terms of unpredictable events causing coordination breakdowns or the support of coordination service improvement or enhancement. A similar concept is defined for coordination mechanisms in the context of CSCW (Schmidt and Simone, 1996).

Most of the above properties are typically not featured by middle-agents, as they are not featured by agents in general. In fact, agents are generally supposed to be autonomous, pro-active, situated entities that interact by means of a general-purpose and high-level communication language (Wooldridge and Jennings, 1995a). As a result, for instance, an agent cannot be supposed to be inspectable. In addition, the general purpose acceptance of the agent notion typically puts limits to predictability, specificity and efficiency.

So, the most obvious embodiment of the notion of artifact for agent coordination is represented by a dedicated abstraction, provided at design time by the coordination model and enacted at runtime by the corresponding coordination infrastructure: that is, an inspectable, efficient, specific, predictable and dynamically forgeable coordination medium that could be used by the agent designer to govern the agent interaction space, but also by intelligent agents to perceive, understand and possibly change the overall MAS behaviour.

4.3 Balancing Coordination in MAS Engineering

Another central notion in AT is the dynamic transformation between levels in collaborative activities. Correspondingly, central to MAS coordination is the support for dynamic transformation from co-operation – that is, the subjective coordination level – to co-ordination (that is the objective coordination level), and vice versa. This is particularly relevant in the context of open and dynamic systems, where the environment is frequently subject to change, and collective goals, norms, and organisational rules should adapt accordingly. This form of dynamism is captured by two basic transitions, the *reflection* and the *reification* of coordination, which must be supported dynamically during system execution. These transitions are strictly related to the transformations seen in the AT, and account for:

- **Reification** – in this transition, coordination laws that have been designed and developed in the co-operation stage are reified in coordination media: intelligent agents forge the behaviour of coordination media in order to reflect the social rules established in the co-operation stage, and to be used as artifacts in the co-ordination stage. It is worth noting that coordination media are meant to embed not only the rules promoting cooperation among agents, but also the laws ruling interactions,

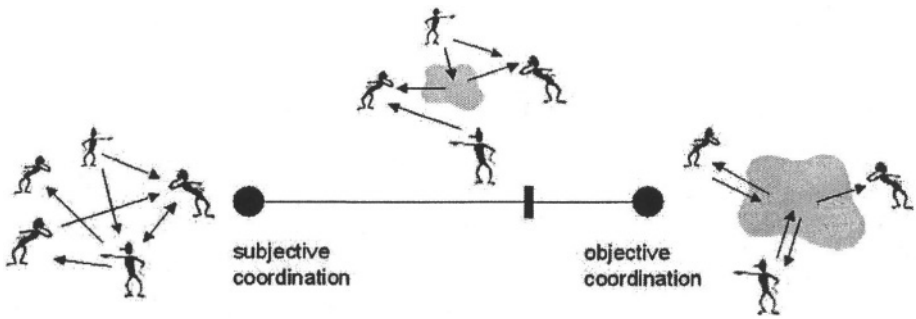


Figure 14.4. The coordination engineering segment: all coordination in agents (purely subjective, left end), all coordination in media (purely objective, right end)

useful to represent also norms and environment constraints, either mediating agent competitive (non cooperative) behaviour, or harnessing self-interested agent behaviours so as to achieve global MAS goals without affecting agent autonomy.

- **Reflection** – in this transition, the behaviour of the coordination media deployed in co-ordination stage is inspected and possibly understood. Agents can retrieve the coordination laws underlying medium behaviour, and relate them to the history of MAS evolution, in order to either evolve them according to changes in coordination policies or in environmental conditions, or learn how to exploit the artifacts in a more effective and efficient way.

The role of coordination artifacts (and correspondingly of coordination infrastructures enacting them) is then central to the engineering of MAS, since they make it possible to balance dynamically subjective and objective coordination, providing the tools to establish at runtime the distribution of the burden coordination between media and agents. As a useful picture, we can draw an imaginary “coordination engineering segment” (see Figure 14.4), whose extremes represent the two opposite situations where on the one side (left in Figure 14.4) all coordination activity are carried on by agents, and on the other side (right in Figure 14.4) all the coordination burden is charged upon the media provided by the infrastructure. Conceptually, the main issue here is to provide the means to devise out at design time where the best “coordination engineering point” of the MAS lays in the coordination segment, that is, the best distribution of coordination activities between subjective to objective orientation, and then to move the coordination point of MAS at execution time, possibly in the follow-up of changes in the MAS environment, to tune the system’s performance, or to modify its behaviour. The position of the point depends on both, the co-

ordination scenario taken into account and the dynamics inside that scenario: the more automation/prescription is required and the more well-defined social rules are (such as for workflow systems), the more coordination knowledge can be represented and enacted through the coordination media. Accordingly, the less possible (or feasible) it is to clearly identify collective rules and constraints in the coordination context, the more the individual agents are to be charged with the coordination burden. As a result, automatable activities are carried out by specialised and efficient coordination artifacts provided by the infrastructure, whilst activities requiring intelligent deliberative capabilities are assigned to intelligent agents. Indeed, the capability of balancing task automation and cooperation flexibly is among the most important requirements for state-of-the-art systems for workflow management, supply chain management, and CSCW (Dayal et al., 2001; Nutt, 1996). The ability to change the “engineering point” of coordination dynamically is also of special importance for open MAS, where the environment can unpredictably change, and the overall structure and functionality of the system may evolve in time.

The above considerations lead to some additional requirements for coordination infrastructures. In particular, in order to support these capabilities, coordination infrastructures should provide the means (languages and tools) for enabling coordination reflection (from objective to subjective transition), to inspect the coordination laws defining medium behaviour, and coordination reification (from subjective to objective transition), defining/programming the behaviour of the coordination media.

5. Engineering MAS with Coordination Infrastructures

5.1 Impact on AOSE Methodologies

The availability of coordination infrastructures has a considerable impact on the process of MAS engineering, and therefore should play a significant role within agent-oriented methodologies. As already mentioned, infrastructures impact on both the final stages of the engineering process (development and deployment) as well as on the analysis and design stages, by means of the abstractions provided by the infrastructure model to represent the environment and to support coordination and organisation.

It is not without reason that AT, which we use as a meta-model to frame the basic elements of a MAS infrastructure model, is primarily used as an analytical tool for understanding collaborative work in complex organisational contexts, and as a design tool to improve them. In such contexts, AT makes it possible to face the complexity of the social activities by clearly separating individual and collective activities, and then by clearly identifying and designing the artifacts required to support both of them. Here we are interested in partic-

ular in artifacts supporting social activities, which we denoted as coordination artifacts.

Along this line, we can devise a correspondence between the levels identified by AT for collaborative activities – co-construction, co-operation and co-ordination – and the engineering stages as typically found in (agent-oriented) software engineering methodologies, i.e., analysis, design, development and runtime. Generally speaking, individual and social tasks are identified and described in the analysis and design stages of these methodologies (Omicini, 2001; Zambonelli et al., 2001a). Individual tasks are typically associated with one specific competence of the system, related to the need to affect a specific portion of the environment and carry out some simple task. Each agent in the system is assigned to one or more individual tasks, and assumes full responsibility for their correct and timely completion. From an organisational perspective, this corresponds to assigning each agent a specific role in the organisation. Conversely, social tasks represent the global responsibilities of the agent system. In order to carry out such tasks, several possibly heterogeneous competences usually need to be combined. The design of social tasks leads to the identification of global *social laws* that have to be respected and/or enforced by the society of agents, to enable the society itself to function properly and in accordance with the expected global behaviour (Zambonelli et al., 2001a).

Given this picture, it is possible to identify a correspondence between the analysis stage (where individual and, in particular, social tasks are identified) and the co-construction level, where the social objectives of the activities are shaped. Then, the identification of the social laws required to achieve the social tasks can be seen as a first step in the co-operation level. This level roughly corresponds to the design and development stages of the engineering process: coordination artifacts are the abstractions which make it possible to design and develop social tasks. At the co-operation level such artifacts are designed and developed to embody and enact – as governing abstractions provided by the infrastructure – the social laws and norms previously identified. Finally, the deployment and runtime stages correspond to the co-ordination level, when the coordination artifacts are instantiated and exploited.

A relevant aspect that it is worth to be pointed out here is that, in the case of AT, the three levels are distinct analytical moments that can be applied continuously, since a collaboration activity is considered to be *continuously* under development, given the intrinsic openness of the environment and the dynamism of organisations. Then, the infrastructure can play a fundamental role not only in providing abstractions and means for the individual engineering stages, but also to support the dynamism between these stages, *continuously*, promoting a form of *online engineering* – a process that appears as unavoidable for the engineering of complex open system (Fredriksson et al., 2003) (see chapter 18).

5.2 Coordination, Organisation and Security in the Same Engineering Context

In the context of MAS, organisation and coordination are strictly related and interdependent issues, and so MAS coordination infrastructures have a fundamental engineering role also in MAS organisation (Omicini and Ricci, 2003),

Generally speaking, *organisation* mainly deals with the structure and the long-term relationships between the components of a system, while *coordination* mainly concerns the processes and the dynamic interactions between the components of a system – often related to roles that usually frame agents in the structure/pattern of system organisation. In any case, both organisation and coordination concern and affect the way in which agents interact with each other, so that conceiving and representing them in the same framework is likely to provide several advantages. Conceptual economy is obviously the first benefit: for instance, the notion of role, usually introduced by organisational models, typically constrains agent actions, which is one of the corner-stones of coordination. Also, a common framework is the most obvious way to consistently support adaptation and evolution of organisation and coordination within an agent society: for instance, by managing explicitly the dependencies between the changes in the organisational settings (such as removal of a role, or changes in its capabilities in terms of interaction protocols) and the related effects on coordination activities. Even more, there are system aspects that can be modelled and engineered in their complex articulation only by considering organisation and coordination settings at the same time: security and electronic institutions are well-known examples. In particular, the multiple aspects related to the security issue in MAS can be tackled in a coherent and satisfactory framework only by covering the whole spectrum that ranges from organisation – with issues related to system structures and relations among the components – to coordination – with issues related to collective processes. Facing security modelling and engineering within this range increases system conceptual integrity, by promoting the reuse of abstractions such as roles, permissions, and societies – which have already proved to be effective in the context of organisation and coordination – in order to enforce complex and dynamic security policies.

Even though the need for run-time liveness of design abstractions supported by the MAS infrastructure follows from basic system engineering considerations, it has an impact on the engineering of intelligent systems (Omicini, 2001). When dealing with MAS organisation abstractions, their liveness allows in principle to dynamically inspect and, possibly, change or adapt it. This is obviously useful for promoting human activities over systems such as monitoring and incremental evolution: however, when dealing with intelligent

systems, the liveness of (organisation/coordination) abstractions is particularly relevant since the properties they embody can be in principle made available not only to humans, but also to intelligent agents. This clearly promotes self-reconfiguration and self-adaptation of intelligent systems: in fact, once an intelligent agent is enabled to inspect the social structure, and allowed to change it, it may reason about the organisation, make inferences, and possibly plan its evolution, for instance to fix some undesired behaviour, or to adapt to environmental changes (Omicini and Ricci, 2003).

Summing up, it is both possible and useful to conceive a MAS infrastructure that supports the modelling and enactment of organisation aspects in synergy with the coordination ones, by keeping the abstractions alive throughout the whole engineering process: that is, by providing MAS engineers with design abstractions also suitable for organisations (such as the notions of role, society, group) and then enabling their management (construction, inspection, adaptation) at both development and execution time. This synergy makes it possible to model and enact coordination activities taking into account the organisation context where they take place, characterised by some structure – in terms of roles, groups, or societies – and organisation rules, such as access control policies. Agents participate to social activities always by virtue of their position (roles) inside the organisation, which define what kind of coordination artifacts they can access and use, and what kind of actions they are allowed (or forbidden) to do on them.

As an example, introduced in (Omicini, 2002), the *Agent Coordination Context* (ACC) abstraction is an infrastructural notion suitable for the integration of organisation issues in a coordination context, especially in the case of artifact-based coordination infrastructure. The ACC notion is meant to model and enact agent position inside an organisational context acting as its environment, so as to define and constrain the agent actions on resources, in this case coordination artifacts (Omicini et al., 2003). Therefore, it is possible to conceive a MAS infrastructure which fruitfully adopts ACC to model and rule agent presence inside the organisation, and, more specifically, agent participation to social activities; this participation includes accessing and using the coordination artifacts as part of organisation resources.

6. An Example of a Coordination Infrastructure

TuCSon is an example of coordination infrastructure for MAS designed according the principles described in previous sections. Figure 14.5 gives a layered perspective of the infrastructure architecture, with organisation and coordination layer in evidence.

TuCSon provides services for the specification and enactment of coordination in MAS (Omicini and Zambonelli, 1999), according to the coordination

as a service approach. Coordination services are embodied in *tuple centres*, that are design/runtime coordination abstractions provided to agents by the infrastructure in order to enable and govern their interaction (Omicini and Denti, 2001). More precisely, tuple centres are *programmable* tuple spaces (Omicini and Denti, 2001), that is, sort of reactive logic-based blackboards; agents interact by writing, reading, and consuming *tuples* – ordered collections of heterogeneous information chunks – to/from tuple centres via simple communication operations (*out*, *rd*, *in*) which access tuples associatively. While the behaviour of a tuple space in response to communication events is fixed and pre-defined by the model, the behaviour of a tuple centre can be tailored to the application needs by defining a suitable set of *specification tuples*, which define how a tuple centre should react to incoming/outgoing communication events, and determine the coordination laws embodied by tuple centres. Tuple centres then can be seen as general-purpose customisable coordination artifacts, whose behaviour can be dynamically specified, forged and adapted so as to automate the co-ordination stage among agents using such artifacts.

The basic infrastructure model is currently being extended to support a role-based organisation model (Omicini and Ricci, 2003). This extension is realised by embodying the ACC notion as first class runtime abstraction (Omicini, 2002). In order to join dynamically a specific organisation, an agent must negotiate and obtain an ACC, as a private interface to access and use the tuple centres of the organisation. The actions enabled by the ACC depend on the active roles the agent is playing inside the organisation.

6.1 Balancing Coordination with TuCSon

In the case of TuCSon, the capability of balancing coordination between subjective and objective as discussed in section 4 is achieved by means of the tuple centre model, and the tools provided by the infrastructure. The coordination laws that define the behaviour of the coordination media (tuple centres) expressed as specification tuples can be inspected and changed dynamically by human and artificial agents by means of specific tools. We are verifying the effectiveness of this approach in scenarios such as pervasive computing – to engineer the social intelligence as required by smart environments – and inter-organisational workflow management systems (Ricci et al., 2002). In the last context, for instance, tuple centres have been used to play the role of the workflow engines, and workflow rules have been expressed as coordination laws embedded within tuple centres. Each workflow engine (mapped onto a tuple centre) acts then as a coordination artifact providing fluid coordination of the individual tasks executed autonomously by human and artificial agents. So, (i) workflow rules are inspectable by accessing the specification tuples embedded in tuple centres (reflection stage); (ii) workflow rules are modifiable at runtime

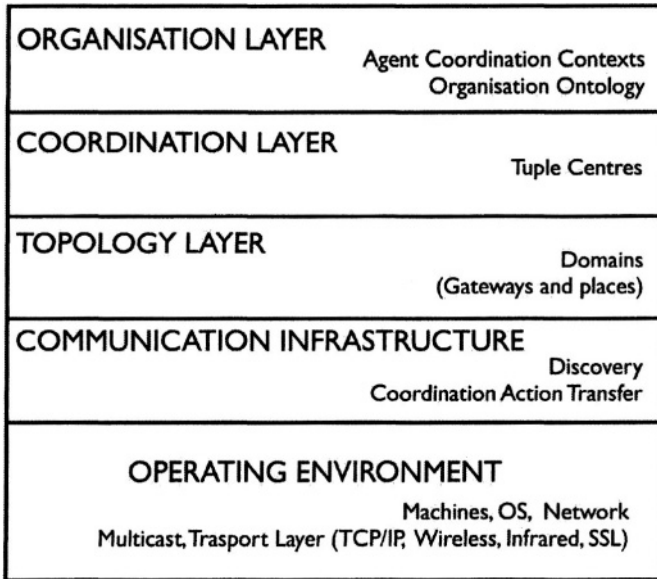


Figure 14.5. The layered architecture of a coordination infrastructure: the TuCSon case

– as a consequence of unexpected exceptions, or changes in the business environment – by changing the specification tuples within tuple centres (reification stage); and (iii) multiple workflow engines (tuple centres) can be exploited, spread over the infrastructure nodes, so as to distribute the coordination workload reflecting a multi-centric view of coordination (Omicini and Ossowski, 2003).

7. Discussion

In this chapter, we provided a brief overview over current conceptualisations, models and support infrastructures for coordination in MAS. We motivated that today’s enabling infrastructures need to be extended so as to allow MAS designers to effectively govern the agent interaction space. Such coordination infrastructures may then become the natural loci for modelling and enacting mechanisms that bias autonomous agent (inter-)action and achieve instrumental behaviour. Drawing from findings in Activity Theory, we put forward the notion of artifact as a step toward a unified framework for coordination, and derived some distinctive properties that a coordination infrastructure should feature. Finally, we provided clues on how these notions can support a principled design process for MAS.

Although the ideas presented in this chapter tackle the problem of coordination infrastructures for MAS engineering mainly at a conceptual level, software

frameworks that adequately support the abstractions that we have put forward will soon be a reality. This, in turn, will facilitate a smooth integration with modern AOSE methodologies and thus allow the full exploitation of the potential of coordination infrastructures in all stages of MAS engineering.

NON TRADITIONAL APPROACHES TO
AGENT-ORIENTED SOFTWARE ENGINEERING

Introduction

Traditionally, software systems are modeled from a mechanical stance, and engineered from a design stance. On the one hand, computer scientists are both burdened and fascinated by the urge to define suitable formal theories of computation, to prove properties of software systems, and to provide formal frameworks for engineering. On the other hand, software engineers are used to analyzing the functionality that a system should exhibit in a top-down way, and to designing software architectures as reliable multi-component machines, capable of providing the required functionality efficiently and predictably. Such traditional stances have been inherited also by researchers in the area of agent-oriented software engineering, as it should have emerged rather clearly from several chapters in the previous parts of the book. Unfortunately, such a traditional way of approaching the engineering of a complex software system may not be necessarily the most appropriate one, especially in the case of very large-systems (i.e., systems composed of a very high number of autonomous components) dived in very dynamic environments.

Modeling and handling systems with a very large number of components in a traditional top-down way can be feasible if such components are not autonomous, i.e., are subject to a single flow of control, and are not influenced by external factors. However, when the activities of these components are autonomous, it is hard, if not conceptually and computationally infeasible, to track them one by one so as to describe precisely the system's behavior in terms of the behavior of its components. In addition, as several software systems are distributed and subject to decentralized control, or possibly embedded in some dynamic environment, an accurate traditional modeling of their behavior (i.e., using traditional logic-based formal models) is simply impossible, as it is predicting and controlling their behavior by design. Software systems of these kind (which are, by definition, multiagent systems, being composed of autonomous and situated components) requires indeed novel modeling and engineering approaches.

From the modeling viewpoint, such systems can only be modeled and described as a whole, in terms of macro-level observable characteristics, just as a chemist describes a gas in terms of macro properties like pressure and temperature.

From the design viewpoint, the lack of micro-level control over the components of a software system and of the environment in which it is dived makes it impossible to obtain a well-defined behavior of the system by top-down design. The challenge for the effective construction of large software systems is to build them in a bottom-up way, starting from simple behaviors of its components and taking care that the interactions between these components and between these components and the environment lead to an overall acceptable global behavior. A reasonable way to face such way of engineering systems is getting inspiration from nature, by reverse engineering and reproducing in computational terms a variety of phenomena – from ant foraging to auto catalytic reactions – in which simple autonomous entities are observed to globally self-organize their behaviors.

The four chapters in this part of the book should provide an broad overview of several approaches that, although very different from each other, all shares the basic underlying idea of adopting novel bottom-up approaches for engineering large-scale self-organizing multiagent systems. In particular:

- Chapter 15, “Engineering Amorphous Computing Systems” by Radhika Nagpal and Marco Mamei, focuses on the problem of engineering the collective behavior of an immense number of simple computational particles by adopting a physically-inspired approach. First, the authors analyze and discuss the potentials of the approach in the context of smart materials and self-assembly. Second, the authors discuss how the same approach can be generalized so as to act as a general-purpose framework to globally coordinate multiagent systems in scenarios such as mobile networks and pervasive computing systems.
- Chapter 16, “Making Self-Organizing Adaptive Multi-Agent Systems Work” by Jean-Pierre George, Bruce Edmonds, Pierre Glize, explores in general terms how emergent behaviors in complex computational systems can be harnessed and fruitfully exploited. In particular, the authors discuss what emergence actually means in the context of multiagent systems, specifically analyze the so called class of adaptive multiagent systems, and discuss an approach for engineering this class of systems.
- Chapter 17, “Engineering Swarming Systems” by Van Parunak and Sven Brueckner, focuses on swarming systems, i.e., multiagent systems that are built by getting inspiration from the behavior of ant colonies and, more generally, from the behavior observed in several classes of social

animals. The chapter introduces swarming systems in general terms, explores why they function, describes the classes of problems for which they are suited for, and outlines some initial principles of an engineering methodology for developing swarming systems.

- Chapter 18, “Online Engineering of Open Computational Systems” by Martin Fredriksson and Rune Gustavsson, explicitly focuses on open computational systems and argues that current agent-oriented software engineering approaches have clear limitations when it comes to their contribution and fulfillment of visions such as ambient intelligence and grid computing. The methodological approach the authors define of on-line engineering is introduced to provide models, methods, and tools to facilitate the necessary transition from programming of abstract machines towards development of grounded physical open computational systems.

Chapter 15

ENGINEERING AMORPHOUS COMPUTING SYSTEMS

Radhika Nagpal and Marco Mamei

Abstract How does one engineer robust collective behavior from the local interactions of immense numbers of unreliable parts? On the one hand, emerging technologies like MEMS are making it possible to assemble systems that incorporate myriad of information-processing units at almost no cost: smart materials, self-assembling structures, vast sensor networks. On the other hand, the plummeting cost of ad-hoc wireless communication is realizing the idea of pervasive computing: the creation of environments saturated with wireless computing devices collectively providing services anytime and everywhere. We discuss organizing principles and programming methodologies for controlling such *amorphous* systems, by combining robust algorithms inspired by nature with computer science techniques for controlling complexity.

1. Introduction

Over the next few decades, emerging technologies will make it possible to assemble systems that incorporate myriad of information-processing units at almost no cost, provided that all the units need not work perfectly and that there is no need to manufacture precise geometrical arrangements or precise interconnections among them. This technology shift requires fundamental changes in methods for constructing and programming computers, and perhaps in our view of computation itself.

Microelectronic mechanical components have become so inexpensive to manufacture that we can anticipate combining logic circuits, microsensors, actuators, and communications devices, integrated on the same tiny chip to produce particles that could be mixed with bulk materials, such as paints, gels, and concrete. Imagine coating bridges or buildings with smart paint that can sense and report on traffic and wind loads and monitor structural integrity of the bridge. A robot, built of millions of tiny programmable modules, could

assemble itself into different shapes, perhaps as a cube for storage and then reconfiguring into a shelter or tool as needed. Already many such novel applications are being envisioned and built (Berlin, 1994; Butler et al., 2001; Cheung et al., 1997; Kahn et al., 1999). Even more striking is the emerging research in biocomputing, that may make it possible to harness the many sensors and actuators in cells and program biological cells to function as drug delivery vehicles or chemical factories for the assembly of nanoscale structures (Weiss, 2001). Pervasive computing and sensor networks are creating massive distributed systems at a different scale, from remote habitat monitoring to smart buildings and smart cars (Mamei et al., 2003a; Priyantha et al., 2000).

These novel computational environments pose significant challenges, beyond just the manufacturing of parts. Digital computers have always been constructed to behave as precise arrangements of reliable parts, and almost all techniques for organizing computations depend upon this precision and reliability. We can envision producing and deploying vast quantities of individual computing elements – whether microfabricated particles or engineered cells or wireless sensors – but we have few ideas for programming them effectively.

The opportunity to exploit these new technologies poses a broad conceptual challenge, the challenge of *amorphous computing* (Abelson et al., 2000):

- 1 How does one engineer robust collective behavior from the cooperation of immense numbers of unreliable parts that are interconnected in local, irregular, and time-varying ways?
- 2 How does one translate prespecified global goals into the local interactions of vast numbers of parts?

The critical task is to identify appropriate organizing principles and programming methodologies for controlling such systems. Hints for how to design robust collective behavior may come from natural systems, such as biology. The growth of form in organisms demonstrates that well-defined shapes and functional structures can develop through the interaction of cells under the control of a genetic program, even though the precise arrangements and numbers of the individual cells are variable. At the same time, as engineers, we must learn to construct systems so that they end up organized to behave as we *a priori* intend, not merely as they happen to evolve. Therefore a critical piece is to develop programming methodologies, and *languages*, that allow us to combine these robust organizational principles to achieve the global goals we want.

In this article we describe work that has been done as part of the amorphous computing effort to address these challenges. Section 2 presents the amorphous computing model, section 3 discusses how developmental biology can provide inspiration for robust algorithms and section 4 presents examples of how we

can combine these algorithms into programming languages. In section 5 we discuss the relationship with pervasive computing and show how similar methods have been developed elsewhere to coordinate behavior in mobile networks of agents (Mamei et al., 2003a).

2. The Amorphous Computing Model

An amorphous computer consists of massive numbers of locally-interacting and identically-programmed computing agents, embedded in space. We can model this as a collection of “computational particles” sprinkled randomly on a surface or mixed throughout a volume.

The individual agents have limited resources, limited reliability and local information. The agents are all programmed identically, although each agent executes its program autonomously and has means for storing local state and generating random numbers. Each agent can communicate with a only a few nearby neighbors. In amorphous systems of microfabricated components, the agents might communicate via short-distance radio or through the substrate itself; bioengineered cells might communicate by chemical means. For our purposes here, we assume that there is a communication radius r , which is large compared with size of individual agents and small compared with the size of the entire area, and that two agents can communicate if they are within distance r . The agents can also sense and affect the environment locally.

In many ways the massively parallel nature of an amorphous computer resembles, and takes inspiration from, models such as cellular automata. However it presents a significantly different challenge because the mechanisms must be independent of the detailed configuration of the agents. We assume that access to centralized sources of information is limited, whether it be global clocks or globally-accessible beacons for triangulating position. Rather the goal is for the agents to self-organize global information as necessary. The agents are not synchronized, although we assume that they compute at similar speeds, since they are all fabricated by the same process. The agents have no *a priori* knowledge of global position or orientation; however some agents may be started in a special initial state. The agents are possibly faulty, and are can die or be replaced at any moment. The individual agents may be mobile, but in many of the examples here we assume that they have fixed location and are randomly distributed on a two-dimensional plane.

3. Developmental Biology as an Inspiration

Biological systems regularly achieve coherent, reliable and complex behavior from the cooperation of large numbers of identically programmed agents. One of the most fascinating examples is embryogenesis. Cells with identical DNA, cooperate to form incredibly complex structures from a nearly homo-

geneous egg, with incredible precision and reliability in the face of constantly dying and growing parts (Wolpert, 1998). There is a plethora of examples of regulation in different organisms, that can compensate for large variations in cell size, cell numbers, cell division rates and development time (Day and Lawrence, 2000). Even after development, organisms such as the starfish, retain incredible abilities for self-repair and regeneration.

These examples hint at powerful underlying mechanisms that can adapt to variation, while maintaining constraints that may be geometric, topological or functional. Studies of developmental biology can form an important source of inspiration – not only for mechanisms, but also for the kind of robustness that is achievable.

Morphogen Gradients. One example of a mechanism common throughout development is the use of gradients of morphogens to determine positional information and polarity. In the *Drosophila* embryo, cells at one end of the embryo emit a morphogen (protein) that diffuses along the length of the embryo. The concentration of this morphogen is used by other undifferentiated cells to determine whether they lie in the head, thorax or abdominal regions (Lawrence, 1992). Different morphogens are used for determining the dorsal-ventral axis, wing and limb development, and even leg bristle polarity. Gradients of morphogens are believed to play an important role in providing position and polarity information in many different organisms, and even in regeneration (Wolpert, 1998).

We can emulate the concept of a morphogen gradient using a simple agent program. An initial “source” agent, chosen by a cue from the environment or by generating a random value, creates a gradient by sending a message to its local neighborhood with the morphogen name and a value of zero. The neighboring agents forward the message to their neighbors with the value incremented by one and so on, until the morphogen has propagated through the entire population. Each agent stores and forwards only the minimum value it has heard for a particular morphogen name, thus the morphogen value represents the shortest path from the source. The value provides an estimate of distance from the source: a point reached in n steps will be roughly distance nr away. The quality of this estimate depends on the density of the agents and can be theoretically predicted for random distributions (Nagpal, 2001).

This very simple program can be used in powerful ways.

- 1 **Regions and Polarity:** By limiting the maximum value of a morphogen, one can create regions of controlled size. The morphogen can also be used to provide a sense of local orientation; an agent can compare values in its local neighborhood to determine the direction towards or away from the source.

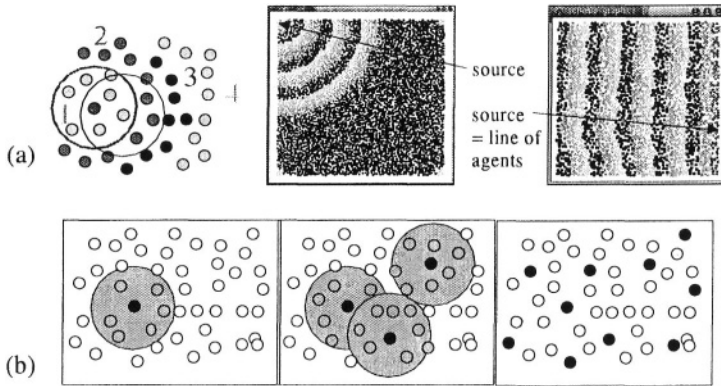


Figure 15.1. (a) Emulating morphogen gradients; gradients created by point and line sources
(b) Spacing created by lateral inhibition

- 2 **Spatial Patterning:** More than one agent could be the source for the same morphogen, in which case the morphogen value reflects the shortest distance to *any* of the sources. Thus, if a single agent emits a morphogen then the value increases as one moves radially away from the agent, but if a line of agents emits a morphogen then the value increases as one moves perpendicularly away from the line. Complex spatial patterning can be created by positioning the sources, without any change to the agent program.
- 3 **Selective Propagation:** The agent program can be modified such that agents selectively choose which morphogens to propagate. Thus agents in particular state can act as barriers to specific morphogens, or as obstacles around which the morphogen must travel. Similarly morphogens can be limited to propagate only with certain spatial regions.
- 4 **Active Morphogens:** We can allow the source agent to constantly produce a morphogen message, and have the morphogen value stored by any agent lose significance if not constantly reinforced. The result is that the morphogen values adapt as agents, or sources, appear and disappear.

These are just a few of the ways in which the morphogen gradients can be used. This simple agent program is the basis of many different amorphous computing algorithms for self-organizing coordinate systems, distributed storage, and ad-hoc routing. In section 5, we see that the same idea appears in many different forms elsewhere.

Lateral Inhibition. While morphogens produce a sense of distance and orientation, lateral inhibition is believed to produce regularly spaced patterns in

many different organisms. For example, in the *Drosophila*, epidermal cells on the leg can produce bristles, however not all cells grow bristles. The bristles are regularly spaced with a minimum distance between them. (Lawrence, 1992) describes the mechanism by which this is achieved: when a cell produces a bristle, it also secretes a short-range morphogen that inhibits nearby cells from growing bristles. An uninhibited cells will eventually attempt to produce a bristle. The result is bristles appear throughout the leg surface but never too close. Lateral inhibition is believed to play a role in creating uniform spacing in many different settings, from the spacing of hair on human skin, to the regular crystal like spacing of ommatidia in the *Drosophila* eye.

Again this mechanism can be emulated by a simple agent program. An agent picks a random number within a range L , and counts down. If it reaches zero without being interrupted, then it becomes a leader (grows a bristle) and sends out an inhibition message to all its neighbors within the distance r . If an agent hears an inhibition message, then it no longer counts down to become a leader but instead becomes a follower. The process ends after L steps, with all agents as leaders or followers.

The result of this very simple local rule is that *leaders emerge with a spacing of r to $2r$ apart, throughout the surface*. If we extend the inhibition to travel h hops distance, then the spacing between leaders increases to hr to $2hr$. To see why this is true, consider an agent that becomes a leader. It could only have done so because no other leader was within distance r inhibiting it; and once it becomes a leader it inhibits leaders forming within distance r . At the same time, an agent that is not inhibited continues to count down and eventually becomes a leader. This guarantees that every agent is within distance r from some leader, and with high probability no leaders are closer than r . The spacing is not perfect because two neighboring agents may choose the same random number and reach zero at the same time. Therefore the range L is chosen to make the probability of such collisions low. Nagpal and Coore have shown that the behavior of this algorithm can be analyzed theoretically, even for asynchronous agents and unreliable agents (Nagpal and Coore, 1998). This is a valuable algorithm in an amorphous computer, and can be used for spontaneously electing leaders, self-configuring hierarchical routing and graph coloring.

Robust Primitives. Morphogen gradients and lateral inhibition are well-matched to the amorphous setting because the gross phenomena of diffusion and spacing are insensitive to the precise arrangement of the individual agents, so long as the distribution is reasonably dense. In addition, if individual agents do not function, or stop broadcasting, the result will not change very much, so long as there are sufficiently many agents. Many phenomena exist in mul-

ticellular systems, from quorum sensing to programmed cell death, that can provide inspiration for robust multiagent algorithms.

At the same time, it is extremely important to be able to analyze the behavior of these algorithms, so that we have a solid ground to build on top of. We can theoretically analyze the behavior of both algorithms, using techniques from distributed graph algorithm analysis and geometric analysis. For example, the morphogen algorithm can be thought of as computing a breadth first search tree, while the lateral inhibition algorithm is a computing a maximum independent set of nodes (Lynch, 1996). The spatial locality of communication gives us a relation from tree depth to distance, and maximum independent set to uniform spacing.

A key difference however, is that rather than produce a perfect answer from a perfect graph, as is typical in distributed algorithms, these algorithms aim to provide a *good-enough answer* with high probability – good enough estimates of distance and direction, and good enough spacing. This allows the agent behavior to be simple, scalable, and tolerant to variation. Looking to biology may provide insights for new approaches to fault tolerance. Traditionally, one seeks to obtain correct results despite unreliable parts. However it seems awkward to describe mechanisms such as embryonic development as producing a “right” organism by correcting bad parts and broken communications. In the amorphous regime, getting the right answer may be the wrong idea. Instead, the question is how do we structure systems so we get acceptable answers, with high probability, even in the face of unreliability.

4. Towards Programming Languages

While biology may provide a means for thinking about organizing local behavior robustly, computer science can provide tools for managing complexity. One such tool is a *programming language*. The ability to think and describe goals in terms of high-level abstractions, make possible a complexity that is almost inconceivable to generate by manipulating 1s and 0s. Yet the final computation does happen as bits, and the *compiler* translates from a language that is natural for expressing how to do something, to a low-level execution model that a computer can interpret (Abelson, 1996).

In the amorphous computing setting the goal is similar – we would like to be able to translate complex global goals into local behavior, but in such a way that the translation is not mysterious and not hand crafted for each goal – in other words, a global-to-local compiler. In this section we describe two programming languages, aimed at pattern formation and self-assembly. The global shape or pattern is described as a program in terms of abstract entities, which is compiled to produce the behavior of an agent, such that the identically-programmed agents organize into the prespecified goal.

```
(define-growing-point (make-red-line length)
  (material red-poly)
  (size 1)
  (tropism (and (away-from red-pheromone)
                (keep-constant pheromone-1)
                (keep-constant pheromone-2)))
  (actions
   (secrete 2 red-pheromone)
   (when ((< length 1) (terminate))
     (default
      (propagate (- length 1))))))
```

Figure 15.2. A program in GPL. This procedure generates a line of specified length that attempts to follow constant values of pheromones 1 and 2

Growing Point Language. Coore developed a language for pattern formation on an amorphous computer (Coore, 1999). The growing point language (GPL) can be used to specify topological patterns consisting of lines of various thickness, such as those specifying the interconnect of an electronic circuit. The specification is compiled into an agent program. Initially the agents start out with identical state except for a few agents. As a result of executing the program, the agents “differentiate” into components of the pattern.

The language represents processes in terms of the botanical metaphor of “growing points” and pheromones. A pheromone is the same as a morphogen with a limited range. A growing point is a locus of activity that modifies the states of agents as it passes through, and it can respond to the gradient of a morphogen by moving towards lower, higher or similar values of morphogens. A pattern is created by writing a program in terms of *abstract entities*: growing points that lay down materials, materials that secrete pheromones, and tropisms that govern the trajectory of the growing point. However, at the level of the agent these high level concepts translate to a set of simple local rules. For example a growing point is simply a piece of state at an agent. The agent collects values of morphogens from its neighbors and uses those value to locally compute which neighboring agent to pass the growing point to. The next agent then repeats the same process to determine where to send the growing point next.

Figure 15.2 shows a fragment of a program written in the growing-point language: A growing point process called *make-red-line*, takes one parameter called *length*. This growing point “grows” material called *red-poly* in a band of size 1. This implies that each agent it moves through sets a state bit that will identify the agent as *red-poly*. The growing point moves according to a tropism that directs it away from areas of higher concentration of *red-pheromone*, in such a way that the concentrations of *pheromone-1* and *pheromone-2* are kept constant. All agents that are *red-poly* secrete

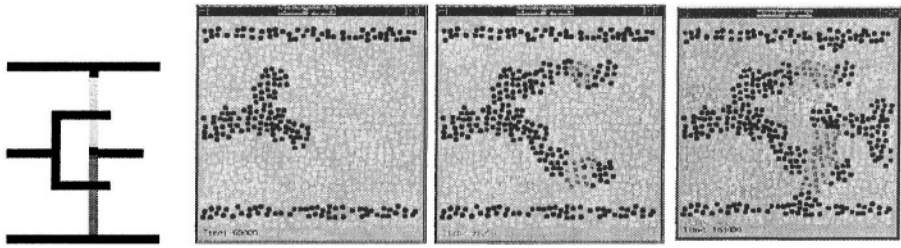
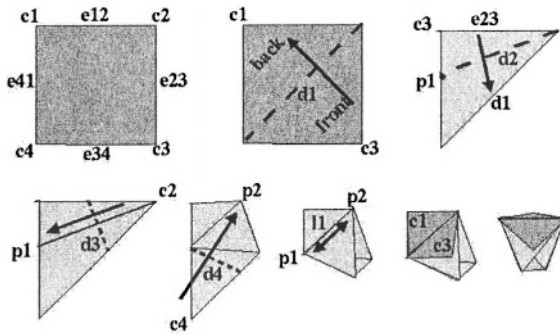


Figure 15.3. The amorphous surface differentiating to create the inverter pattern. All agents execute the same program, which is compiled directly from the GPL specification of the pattern on the left

red-pheromone; consequently, the growing point will tend to move away from the material it has already laid down. The growing point stops when the correct length line has been grown. This procedure is part of a larger GPL program that generates the pattern on the right. This pattern is a caricature of the layout of a CMOS inverter, where the different colored regions represent structures in the different layers of standard CMOS technology: metal, polysilicon and diffusion. Figure 15.3 shows the agents differentiating to create the inverter pattern. The agents that are part of the top blue rail emit pheromone 1 and the bottom rail emits pheromone 2, thus the code fragment represents the method by which the first red line is drawn parallel to these two rails and away from the edge. The entire program that specifies the shape is only a few paragraphs long, and the resulting state machine for the individual agents requires only about twenty states.

Programmable Self-Assembly. Nagpal developed a language for shape formation on a simulated foldable sheet (Nagpal, 2002). In this case the two dimensional surface of agents represents a sheet with a single layer of randomly and densely distributed agents; a set of agents in a line can coordinate to fold the sheet along that line. The folding is modeled abstractly by the simulator, but is inspired by epithelial tissues where a line of epithelial cells can deform to cause the entire sheet to fold along that line; this is believed to be the basis of neural tube formation during embryogenesis (Wolpert, 1998). One could imagine building a programmable reconfigurable sheet composed of such flexible agents.

The shape is specified as folding construction on a continuous sheet, using a language called the Origami Shape Language (OSL). The language is based on a set of geometry axioms, described by Huzita to capture the mathematics behind origami paper-folding (Huzita and Scimemi, 1989). A large class of flat folded shapes and line patterns can be constructed using these axioms. OSL



```
;; OSL Cup program
;;-----
(define d1 (axiom2 c3 c1))
(define front (create-region c3 d1))
(define back (create-region c1 d1))
(execute-fold d1 apical c3)

(define d2 (axiom3 e23 d1))
(define p1 (intersect d2 e34))
(define d3 (axiom2 c2 p1))
(execute-fold d3 apical c2)

(define p2 (intersect d3 e23))
(define d4 (axiom2 c4 p2))
(execute-fold d4 apical c4)

(define l1 (axiom1 p1 p2))
(within-region front (execute-fold l1 apical c3))
(within-region back (execute-fold l1 basal c1))
```

Figure 15.4. OSL Program for folding a cup

builds on these geometry axioms, but also adds concepts such as naming and regions.

The interesting thing about this specification is that it is abstract – there is no notion of morphogens, coordination or even agents. Rather the programmer thinks in terms of a continuous sheet. The agent program is automatically compiled from this description and is composed from a small set of primitives: morphogens, neighborhood query, cell-to-cell contact, polarity inversion and flexible folding.

Figure 15.4 shows a diagram for constructing a cup from a blank square sheet of paper, and the corresponding OSL program. The basic elements of the language are points, lines and regions. Initially, the sheet starts out with four

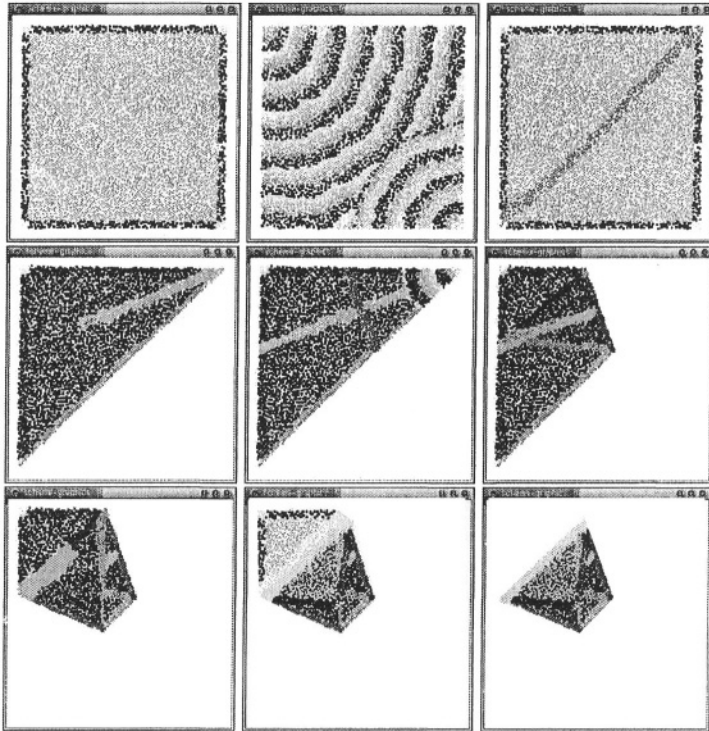
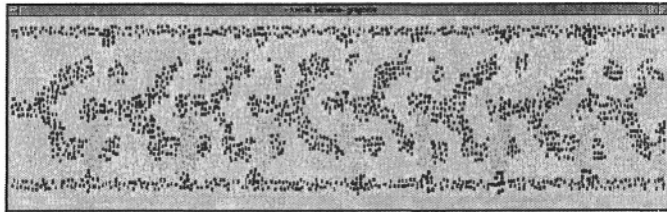
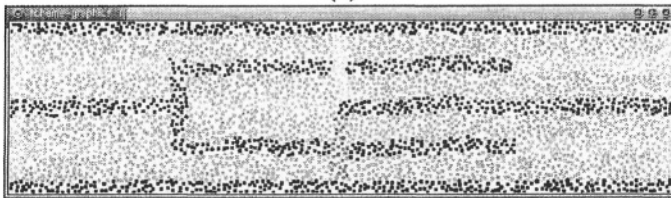


Figure 15.5. Simulation images from folding a cup



(a)



(b)

Figure 15.6. Inverter pattern created by (a) GPL (b) OSL when run on a longer sheet

corner points (c_1-c_4) and four edge lines ($e_{12}-e_{41}$). The axioms describe how to generate new lines and points from an existing set of lines and points, purely through folding and unfolding paper. For example, the first operation constructs the diagonal d_1 from the points c_1 and c_2 by using axiom 2; axiom 2 folds the sheet so that c_1 lies on c_2 and then unfolds the sheet to create a line. The sheet can be permanently folded flat along a line, hence the structures created by OSL are flat, but layered. Lines can be used to create regions and regions can be used to restrict folds.

Figure 15.5 shows a programmable sheet differentiating to fold into a cup. Initially the surface is mostly homogeneous, with only the agents on the border having special local state. When the agent program is executed by all the agents in the sheet, the sheet is configured into the desired shape. The overall *global* view of this process is very close to what the diagram of the continuous sheet suggests. This is because each global operation is translated into a local agent behavior. But instead of folding, the geometry is emulated using the biologically-inspired primitives. For example in order to implement the first line creation, the agents belonging to c_1 and c_2 create two distinct morphogens. The remaining agents test if the morphogen values are equal; if so then they lie on the new line. This is the local rule corresponding to axiom 2. Morphogens also serve as a form of barrier synchronization, so that agents can determine when it is safe to move on to the next fold operation. Selective propagation of morphogens is used to create regions and confine operations within regions. Each axiom translates into a set of local rules, and the program translates to a sequence of rules.

The Power of Programming Languages. The power of the programming language approach is that it allows us to take advantage of traditional computer science techniques for managing complexity, while relying on biological models for achieving robustness at the local level. The global-to-local compilation confers many advantages: (i) we can reason about the classes of structures that can and cannot be generated by analyzing the expressiveness of the language; (ii) the primitives themselves can be made robust by relying on mechanisms inspired by biological systems; (iii) the analysis of a complex system becomes tractable because it is built in understood ways from smaller parts; and (iv) the high level language makes it possible to *easily* specify complex behavior, without worrying about the millions of parts that are involved.

For example, Coore proved that GPL can generate any prespecified planar graph pattern, up to connection topology, on an amorphous computer. Similarly, OSL can generate any 2D Euclidean construction pattern and all flat folded shapes composed of simple folds. These results are based on results from geometry, that have nothing to do with multiagents or self-organization. At the same time we can separately analyze the robustness of primitives such

as morphogens, and how error accumulates when we combine those primitives, so that we can predict what densities and numbers of agents are required to satisfactorily achieve a given high-level goal. Similarly we can analyze time and space complexity. In both languages, the complexity of the agent program is directly proportional to the complexity of the high-level description; by using *procedures* to capture regular patterns and common folding sequences, one can compile more efficient agent programs.

The languages themselves imply certain global properties. For example, the GPL encodes patterns with an *inherent length scale* and can easily describe fractal and space filling structures. The OSL language on the other hand describes structures in a scale-independent manner, by recursively segmenting relative to the original sheet boundary. This results in patterns that scale automatically, and even asymmetrically, without any change to the program. For example when the GPL program for an inverter is executed on a long sheet, it results in a chain of inverters of the same size. In OSL a longer sheet simply stretches the inverter (Figure 15.6). The two languages encode very different properties, that can be derived directly from the choice of high-level language, and result in different local strategies. Insights from these languages can be used to design new languages with similar properties.

So far the work in amorphous computing has focused on languages for pattern and shape formation. However, the desire to achieve global-to-local programming is not unique to amorphous computing. The emerging field of pervasive computing poses the same challenge – vast numbers of computing devices embedded in our everyday environments, need to be programmed so that specific global services result from their coordinated activities. In the rest of this chapter, we discuss how this programming methodology can impact pervasive computing.

5. Pervasive Computing

Consider a scenario a few years hence in which a large city like Boston might have several wireless base stations in every building – a number of nodes in the order of 10^7 . If most of the electrical devices in the buildings and those carried on by people are wirelessly networked too, then the total number of nodes could be as high as 10^{10} . If these nodes communicate peer-to-peer with nearby devices, then one could envision the entire city as connected into a mobile ad-hoc network approximately 10^3 hops in diameter. It is clear that this *pervasive computing* scenario strongly resembles the amorphous computing model, and poses similar challenges, with the significant addition of mobility.

For example, consider the problem of programming a group of agents to coordinate their movements through an environment. Such agents may represent humans carrying wireless PDAs, navigator-augmented cars, or auto-

mous robots. As in the amorphous computing scenario, instead of programming the individual agents behavior directly, we would like to express desired motion patterns in a global way. For example, in a traffic management application, the goal may be to engineer collective behaviors to reduce the traffic. We would like to be able to translate the desired global behavior into the agents' local interactions (e.g., who gives the precedence to whom) (Mamei et al., 2003b). Because of these analogies, we started to look at ideas similar to the ones exploited in amorphous computing to organize the behavior of pervasive computing devices.

Coordinated Movement in Mobile Agents. In our research project at University of Modena and Reggio Emilia, we have used an idea similar to that of morphogen gradients, which we call fields, to drive agent motion patterns. Agents are wirelessly connected in a mobile ad-hoc network (e.g., they are humans carrying on Wi-Fi PDAs) and fields have been modeled by means of distributed data structures, created by an agent, and propagated to its neighbors hop-by-hop. Specifically, we have developed the TOTA (Mamei et al., 2003a) middleware that provides agents with a high-level interface to define and spread these distributed data structures. Each field is defined by means of a content C and a propagation rule P identifying how the field should distribute in the network and how its value should change during the distribution.

Moreover, to take into account the dynamism of mobile networks the spatial structures resulting from field propagation are kept coherent by the TOTA middleware despite network dynamism (see Figure 15.7). From the agents's viewpoint, executing and interacting are basically reduced to injecting fields, perceiving local fields, and acting according to some application-specific policy.

Let us focus on an example, imagine security guards in a museum who move and monitor the museum in a coordinated way; they have to preserve a specified distance, say d , from each other. The security guards can be provided with wireless enabled palm computers, connected through an ad-hoc network and running the TOTA middleware. Each guard's palm (agent) can generate the field in Figure 15.8a that propagates in the surroundings and reaches a minimum value at distance d from the agent. The final shape of this field approaches the distribution shown in Figure 15.8b. Each agent can then sense its immediate neighborhood, looking for the fields generated by all the other guards. It can combine the perceived fields, by computing the minimum value at each point. The result can be considered a field in itself, having minimum points at distance d from other agents. At this point, each agent can just follow downhill this computed virtual field. The field is automatically updated as the agent moves. The result is a globally coordinated movement in which agents reach and maintain an almost regular grid formation (see Figure 15.8c). Following

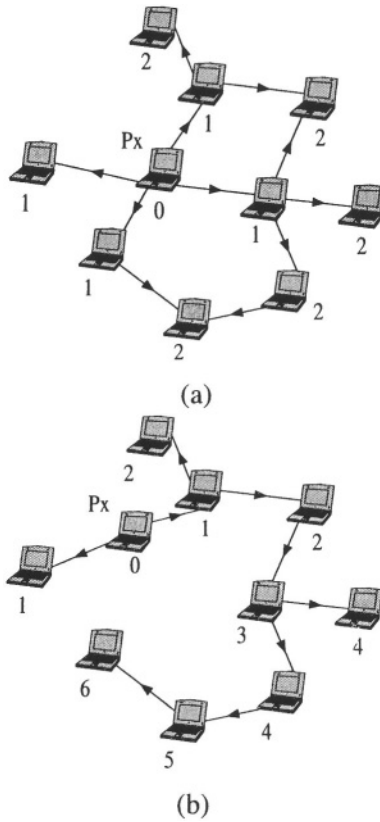


Figure 15.7. TOTA keeps a distributed fields' structure coherent despite dynamic network reconfigurations: (a) A peer P_x propagates a field that increases its value by one at every hop (b) When the field source P_x moves, all fields are updated to take into account the new topology

similar strategies, it is possible to realize a vast number of coordinated motion activities, e.g., have a group of agents to meet somewhere, let them move avoiding the emergence of crowds or queues, let them cooperatively surround a prey, etc. (Mamei et al., 2003b).

It is worth noting that the TOTA approach is adaptive, in that the fields associated to a given motion coordination policy automatically adapt to the environment in which they are propagated. For example, if the application is executed in a building, as long as the rough mobile network topology resembles environmental physical constraints (i.e., no network links through walls), the fields' shape and thus the coordinated motion patterns adapt to the building topology, without any change in the application code. This property resembles the scale-independence property of the OSL language described in section 4.

C = (peerName, val)
 P = ("val" is initialized at d, propagates to all peers, decreasing by 1 in the first d hops, then increasing "val" by 1 for further hops)

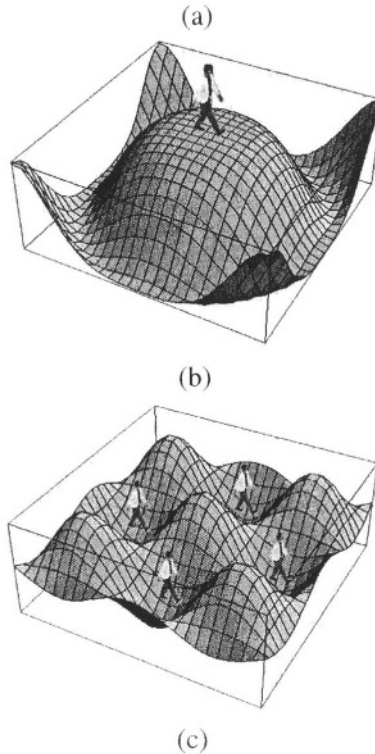


Figure 15.8. (a) Field propagation rule (b) Distribution of a single field (c) Regular formation of peers

Moreover, in TOTA – as in GPL and OSL – agents are not designed in isolation. In the case study, for example, agents achieve the goal of maintaining the formation not because of their own internal algorithm – actually, *they do not even know about any kind of formation* – they just follow downhill the gradient of the fields being propagated, but this allows the system, *as a whole*, to enforce the formation.

Apart from these similarities, there is still a gap between this approach and the GPL and OSL programming languages. In TOTA, in fact, we still have not identified a general methodology to help us identify, given a specific motion pattern to be enforced, which fields have to be defined, how they should be

propagated, and how they should be followed by agents. So this must be coded by hand for each specific application. On the contrary, the availability of a programming language like GPL or OSL, would enable us to specify the *global* motion pattern we would like to achieve and have a compiler to automatically derive suitable fields, propagation rules and agents' algorithm to follow fields. We are confident that such a programming language can be found in the future, and would possibly make the model applicable to a wider class of distributed coordination problems, even beyond motion coordination.

Other Examples. Distributed data structures, like morphogen gradients, driving agent activities, are emerging in many disparate scenarios. The research projects Anthill (Meling et al., 2001) and SwarmLinda (Mendez and Tolksdorf, 2003) both use algorithms based on field-like data structures spread in the network by mobile software agents to enable file sharing in Internet-scale peer-to-peer (P2P) applications. Instead of being propagated in a breadth-first manner, like the morphogen gradients, the agents spread the data structure as they randomly move across the network. As a result paths are created between peers that share similar files, thus enabling a fast content-based navigation in the network of peers.

Similarly, in wireless ad hoc networks, morphogen gradients and fields can be used to design of routing mechanisms. Examples of this can be found in Gradient Routing (Poor, 2001) and in Directed Diffusion (Intanagonwiwat et al., 2000), where peers spread morphogen gradients across the network, so that other packets can reach their intended destination by following downhill the gradient associated with the destinations. Analogous techniques have been also used in (Nagpal et al., 2003) to create a coordinate system over ad-hoc networks. Here nodes evaluate their coordinates by triangulating the distances – expressed by means of morphogen gradients – from elected beacons.

In robotics, the idea of fields driving robots movement is not new. One of the most recent re-issue of this idea, the Electric Field Approach (EFA) (Johansson and Saffiotti, 2001), has been exploited in the control of a team of Sony Aibo legged robots in the RoboCup domain. Following this approach, each Aibo robot builds a field-based representation of the environment from the images captured by its head mounted camera, and decides its movements by examining the fields' gradients of this representation. Similarly, Pheromone Robots (Payton et al., 2002) use field like distributed data structures to drive robot vehicles to achieve useful large-scale results in surveillance, reconnaissance, hazard detection, and path finding.

A modular robot is another example; it is a collection of simple autonomous actuators, with few degrees of freedom, connected with each other. A distributed control algorithm is executed by all the actuators to let the robot assume a global coherent shape or a global coherent motion pattern (i.e., gait).

An interesting proposed approach adopts an idea similar morphogen gradients to control such a robot (Shen et al., 2002). Here, morphogen gradients (called hormones) are created and propagated through the robot. Robots' modules decide how to bend their actuators depending on the locally perceived hormone pattern.

Shifting from physical to virtual movements, the popular videogame "The Sims" (see <http://thesims.ea.com>) exploits sorts of computational fields, called "happiness landscapes" and spread in the virtual city in which characters live, to drive the movements of non-player characters. For instance, if a character is hungry, it perceives and follows a happiness landscape whose peaks correspond to places where food can be found, i.e., a fridge.

The fact that similar notions, such as gradients, are found everywhere, suggests that they are fundamentally suited to these types of environments. However high-level programming languages and global-to-local compilation are rare. In the amorphous computing examples, the programming languages made it possible to easily achieve complex and robust desired behavior. We believe that in these other environments, the invention of appropriate global languages could have a similar far-reaching impact.

5.1 Acknowledgements

The Amorphous Computing project was started by Abelson, Sussman, and Knight, and the work discussed here reflects the contributions of many people in the Group. Support for Amorphous Computing research was provided in part by the Advanced Research Project Agency of the Department of Defense, contract number N00014-96-1-1228, and in part by a grant from the National Science Foundation, Division of Experimental and Integrative Activities, contract number EIA-0130391. Further support was provided by the Italian MIUR and CNR in the "Progetto Strategico IS-MANET, Infrastructures for Mobile ad-hoc Networks."

Chapter 16

MAKING SELF-ORGANISING ADAPTIVE MULTIAGENT SYSTEMS WORK

Towards the Engineering of Emergent Multiagent Systems

Jean-Pierre Georgé, Bruce Edmonds and Pierre Glize

Abstract The context of computational entities is rapidly changing: the development of artificial systems such as the Internet, ubiquitous computing, pervasive computing and autonomic computing mean that these entities have to cope with emergent phenomena arising in their environment. Rather than attempt to eliminate such emergence, we start to explore how this might be deliberately harnessed. That is, address how we might seek to engineer MAS with desirable emergent properties. To do this we discuss what emergence might mean in the context of MAS, and consider a class of such systems: Adaptive MAS (AMAS) that might be used to bring about such emergence. After reviewing the theoretical adequacy of AMAS systems we go on to sketch an approach to making them: namely by focusing the design effort on equipping each agent with responses to the non-cooperative situations it may encounter. This approach is illustrated in a simple but effective flood forecasting system called STAFF. Finally we discuss the expected benefits and difficulties inherent in this approach and the likely way forward.

1. Introduction

The traditional design approach in software engineering requires the designer to have some important initial knowledge: first, the exact purpose of the system, and second, the range of possible situations (e.g., interactions) to which the system may be confronted with in the future. This point of view, which leaves little room for the system's autonomous operation, has been guided by two converging considerations from the beginning of computer science. They are:

- To guarantee in the most formal way possible that the system effectively computes the “right” function (or achieves the stated goal); and

- To optimise memory capacities, computing speed and the very limited perceptive capacities associated to the first computers.

Such an approach imposes an ever-growing design task on the developer resulting from the increase in complexity of the problems being tackled (enabled by the constant increase of computer power and their inter-connectedness). This increasing burden has motivated the invention of many software development techniques, which nevertheless do not solve the underlying problem. That is they only manage to slow down – without stopping – the increase of human effort necessary to deal with ever more complex systems and situations.

This chapter explores an alternative approach which may help deal with this burden. Basically this approach aims to build-in the ability of the agents to learn so that the whole system of agents can adjust itself to what is required based on the feedback it is given. This is in contrast to trying to infer the right behaviour to meet a defined goal. The difference lies in the *knowledge* that the agents have – their own supply of learnt knowledge can (in a system with adequate feedback and interaction) substitute somewhat for the prior knowledge that the designer would otherwise need to have known. The idea is thus simple – the question remains of how to “structure” the agent system so that this adaptation occurs and is sufficiently effective so that the whole system is useful.

The evolution of computer science forces us to consider that it is more and more difficult – if not impossible – to control accurately the activity of increasing complex software systems or even describe completely how they work. As (Horn, 2001) says:

“Even if we could somehow come up with enough skilled people, the complexity is growing beyond human ability to manage it. As computing evolves, the overlapping connections, dependencies, and interacting applications call for administrative decision-making and responses faster than any human can deliver. Pinpointing root causes of failures becomes more difficult, while finding ways of increasing system efficiency generates problems with more variables than any human can hope to solve. Without new approaches, things will only get worse.”

This problem is one that can only get worse. One factor that will ensure this is the development of ubiquitous computing (see chapter 19).

“The Ubiquitous Computing era will have lots of computers sharing each of us. Some of these computers will be the hundreds we may access in the course of a few minutes of Internet browsing. Others will be imbedded in walls, chairs, clothing, light switches, cars – in everything. Ubiquitous Computing is fundamentally characterized by the connection of things in the world with computation. This will take place at a many scales, including the microscopic” (Weiser, 1996).

The growth of ubiquitous computing mirrors that of the Internet – interaction with this is increasingly common in many applications. For this reason you cannot think of having a global control over this kind of application. Moreover, the power of each computer enables us to perform more complex computations – each system can be composed of many interacting entities that are autonomous, heterogeneous and evolutionary (Parunak and Vanderbok, 1997). Finally, the many possibilities for connecting different hardware systems imply that we have to take into account open and heterogeneous environments that are increasingly dynamic.

One way to counter these difficulties is to give more autonomy to the software so as to enable it to adapt itself, as well as it can, to unexpected events. In this case we make the agents more autonomous by allowing them to learn knowledge that the designer does not have. Making machines more autonomous means that the designer can defer some of the design decisions to the software itself. This means that the agents may be able to make better decisions since they have up-to-date information. However this deferring of decision making is not easy to do in a truly distributed way and it is easy to fall into the trap of having assumed something about the context of operation that turns out not to be the case and so not endow the system with an ability to cope with it.

Formal theories can help us to represent and reason about time, space and dynamics of an evolving world. However such an approach is very difficult to make work in many situations, including when:

- The system's environment is very dynamic, making it ineffective to enumerate exhaustively all the situations the system may encounter (or encode it in a suitable decision mechanism), e.g., a real-world robot.
- The system is open, in effect constituted of a shifting number of components (e.g., an e-marketplace).
- The task the system has to achieve is so complex that it is difficult to formulate adequate goals and it is infeasible to check that a design is adequate (e.g., Ubiquitous Computing where the goal is to satisfy the users; but what is "user satisfaction" for a UC system? For a discussion about UC see chapter 19).
- The way a system achieves the task it has been assigned is difficult or even impossible to apprehend in its totality by the designer (e.g., flood or weather forecast: how does the system link all the different parameters during time to give an adequate forecast? See section 4).

If we are to be able to produce artificial systems with the ability to confront really unexpected situations (like the ones resulting from a lack of spatial or

temporal knowledge) then we need to develop a research agenda that leads to a different paradigm of design than the traditional global top-down approach based upon an adequate modelling of the world (Edmonds, 1998).

The more autonomy a system has, the less easy it is for a designer to control it or even predict what it will do. This is both the advantage and the disadvantage of using autonomous agents for practical tasks. When the task is to be undertaken by a group of interacting autonomous agents (whose autonomy includes adaptation) both the advantages and the disadvantages are greatly amplified. Baldly put – it is very difficult to get a bunch of truly autonomous entities to do what you want them to do. It is doubly difficult to get them to cooperate to achieve any collective goal you may have, since the complex and dynamic interactions that could occur are beyond our ability to analyse and predict, in general.

Fortunately we have a rich source of ideas and examples from which to draw – human society. It is the analogy with human (and other animal) actors that makes the agent paradigm useful – otherwise it would be simply a bunch of arbitrary interacting modules with no guide with which to direct their design.

One such approach to controlling such systems of agents is to impose a tight managerial control upon them – effectively reducing their autonomy in order to achieve desired collective results. This can be dubbed the “managerial” approach. It is the approach taken by human managers in firms for whom control is important. An example of this is (Wooldridge and Jennings, 1998) who lists some of the pitfalls of agent development each of which essentially constrain the autonomy of the composite agents in order that a traditional design approach can operate.

However, if one abandons the pure design stance, there are other possibilities. In the human case there are many circumstances where control is less important than the success of the group – in fact, there are some circumstances where *more* control means *less* success. For example, the attempt to control all aspects of production in the Soviet Union with 5-year plans was not as successful as the relatively “anarchic” market economies of the West¹. Similarly in search techniques, the unpredictability of the search landscapes means that “greedy” techniques, where search is directed, can often result in worse results than search which involves more randomness. Basically these other possibilities all impose *some* structure and endow each agent with some communication ability and motivations, thus allowing each agent with more autonomy than in the managerial approach. For example, such agents could join or leave groups dynamically, could entrepreneurially develop novel services for other agents or be an information broker.

¹Considering only this aspect without regard to other parameters (cultural, social, ecological, etc.)

Thus systems of adaptively autonomous agents look like they are a fruitful approach to the understanding and production of useful complex emergent systems. They have the flexibility to implement any desired systems but yet there are some guiding ideas and analogies to guide us in controlling and understanding them. Their adaptivity holds out the hope that we will be able to defer some of the decision making to the system at the decision time.

2. Characterization of Emergence in Synthetic Systems

Research on self-organization (see chapter 17) tries to describe and explain forms, complex patterns and behaviours that arise without an outside organizer. See a special issue of the Royal Society (Royal Society of London, 2003) for an analysis of phenomena observed in physical systems, in nature and in social contexts. One common characteristic of the mechanisms that trigger and create self-organisation is the use of simple rules for the emergence of complex processes.

To be able to use the notion of emergence in artificial systems, some strict rules must be followed. Indeed we must be careful not to introduce, inadvertently, some form of control which would prevent the appearance of emergence. In order to do that we present briefly in this paragraph what is the common agreement about the meaning of emergence in natural systems and what are the deduced rules for artificial ones.

2.1 The Meaning of “Emergence” in Natural Systems

So what allows us to judge of the emergent character of a phenomenon? Instead of pretending to give an exact and exhaustive definition of emergence, we will rather, using definitions extracted from literature, list those properties that seem fundamental to us in order to be able to comprehend in a precise way the notion of emergence. The common inter-related proprieties that let us identify a phenomenon as emergent are:

- 1 The observation of an ostensive phenomenon (it imposes itself to the observer) at the global or macroscopic level. Emergence is defined in terms of the irreducibility of properties associated with a higher level theory to properties associated with components in a lower level theory (Kim, 1997).
- 2 The radical novelty of the phenomenon (it is not visible at the microscopic level and it is impossible to predict it). “On one hand, emergence presupposes appearance of novelty – property, structure, form or function –, and on the other hand, it implies that it is impossible to describe, explain or predict these new phenomena in physical terms from the basic conditions defined on the lower levels” (van de Vijver, 1997).

- 3 The coherence and correlation of the phenomenon (it has an own identity but it is strongly linked to the parts that produce it). “Emergence refers to the arising of novel and coherent structures, patterns, and properties during the process of self-organisation in complex systems” (Goldstein, 1999).

Taken together and applied to software these mean that the emergent phenomenon is some pattern in the outcomes that is meaningfully identified as a relevant phenomenon in its own right, that is, it is a novel, coherent phenomenon that imposes itself on us at the global level. Further that, although it is caused by the detail of the software operation (and its environment), this phenomenon is neither completely predictable from the code nor analysable down to the design of the code and its environment (at least in practice). One consequence of this is that any particular emergent effect can *not* be designed into any set of code as the result of any prior theory (otherwise it would be predictable and analysable to that code). This does not stop considerable elements of design entering in to the making of a system with desirable emergent properties – for example, it may be possible to determine some necessary conditions for the desired phenomenon, but not conditions that are also sufficient. However it does mean that prior design processes are not sufficient on their own, there needs to be some processes of *a posteori* adjustment or evolution as the result of feedback concerning the phenomenon (Bryson, 2001).

2.2 Computational Approach for Emergence

During the last fifteen years, a whole research field slowly appeared around the concept of emergence so as to exploit its so particular characteristics in computational systems (Holland, 1997). “Emergent Computation” (Forrest, 1991; Gilber, 1995) is the term used in general to refer to the research main line on which our work stands.

Our work in this domain during the last decade leads us to give a “technical” definition of emergence in the context of MAS, and therefore with a strong computer science coloration. It is based on three points:

- 1 **The subject.** The goal of a computational system is to realise an adequate function, judged by a relevant user. It is this function (which may evolve during time) that has to emerge.
- 2 **The condition.** This function is emergent if the coding of the system does not depend on the knowledge of this function. This coding has to contain the mechanisms facilitating the adaptation of the system during its coupling with the environment, so as to tend towards an adequate function.

- 3 **The method.** To change the function the system only has to change the organisation of its components. The mechanisms which allow the changes are specified by self-organisation rules providing autonomous guidance to the components' behaviour without any knowledge of the collective function.

So when we design an agent for a MAS, the code of the agent does not contain any knowledge of the collective function we want the MAS to compute. This function has to emerge as a result of adaptation by agents as a result of their interactions in the system and the feedback to the system.

3. An Example of a MAS Technology using Emergence

The AMAS (Adaptive Multi-Agent Systems) approach is described here to show what could be a method for producing MAS with desirable emergent properties. In this approach the adequate collective behaviour is based on the cooperative action of the agents (Caperla et al., 2003a).

The first subsection explains the central role of the system's organisation for its global function. The second one summarises the theoretical adequacy of the approach, namely a justification of the interdependence that can exist between a cooperative local behaviour and the adequacy of the collective global function. The third shows how this theory can be applied to adaptive MAS. We explain the basic behaviour of an agent facing locally uncooperative situation and the consequence at the organisation level.

3.1 Adapt the System by Its Parts

By specifying *a priori* a model for a system that will have to deal with unexpected events, you constrain (maybe inopportunately) the space of possibilities. Since (von Bertalanffy, 1968), many authors have studied systems of different order that cannot be apprehended by studying their parts taken separately: "We may state as characteristic of modern science that this scheme of isolable units acting in one-way causality has proven to be insufficient. Hence the appearance, in all fields of science, of notions like wholeness, holistic, organismic, gestalt, etc., which all signify that, in the last resort, we must think in terms of systems of elements in mutual interaction."

Following this view, we consider that each part of a MAS achieves a partial function. These partial functions will be combined in some way to construct the global function (which is the output of the system). The combination of partial functions is determined by the current organisation of the parts and, in general, the order and manner in which they are combined will matter. Thus by transforming the organisation of the MAS, you change the combination of the partial functions and so you modify the global function. Therefore, this is a mechanism that can be exploited to adapt the system to its environment. A per-

tinent framework within which to build this kind of system is that of the adaptive MAS. As usually meant (Wooldridge, 2002) by “multiagent systems,” we will be referring to systems constituted by several autonomous agents, plunged in a common environment and trying to solve a common task.

3.2 The AMAS Theory

Cooperation was extensively studied in computer science by Axelrod (Axelrod, 1984) and Huberman (Huberman, 1991) for instance. “Everybody will agree that co-operation is in general advantageous for the group of co-operators as a whole, even though it may curb some individual’s freedom.” (Heylighen, 1992). Relevant bio-inspired approaches using cooperation are the Ants Algorithms (Dorigo and Di Caro, 1999) which give efficient results in many domains. In order to show the theoretical improvement coming from cooperation, we have produced the following theorem which describes the relation between cooperation in a system and the resulting functional adequacy of the system.

We demonstrate the **functional adequacy** of a particular class of systems (the class of cooperative internal medium systems) by the fact that the systems of this class are adequate to carry out the task for what they were conceived.

Note, “functional” refers to the “function” the system is producing, in a broad meaning. I.e., what the system is doing, what an observer would qualify as the behaviour of a system. And “adequate” simply means that the system is doing the “right” thing, judged by an observer or the environment. So “functional adequacy” can be seen as “having the appropriate behaviour for the task.”

Theorem *For any functionally adequate system, there is at least a cooperative internal medium system that fulfills an equivalent function in the same environment.*

Note, a *cooperative internal medium system* is a system where no Non-Cooperative Situations (NCS) exist (see section 3.3 for a definition of NCS).

This theorem means that we only have to use (and hence understand) a subset of particular systems (those with cooperative internal mediums) in order to obtain a functionally adequate system in a given environment. We concentrate on a particular class of such systems, those with the following properties (Gleizes et al., 1999):

- It is a cooperative system which is functionally adequate with respect to its environment. Its parts do not “know” the global function it has to realise via adaptation.
- It does not have an explicitly defined goal, rather it acts using its perceptions of the environment as feedback so as to adapt the global function to be adequate. The mechanism of adaptation is for each agent try

and maintain cooperation using its skills, representations of itself, other agents and environment.

- Each part only evaluates whether the changes taking place are cooperative from its point of view - it does not know if these changes are dependent on its own past actions.

The demonstration of the functional adequacy theorem in (Camps et al., 1998) results from the application of the following axiom and the four lemmas. For each of them we have added a short textual explanation.

Axiom *A functionally adequate system has no antinomic activity on its environment.*

Note, *antinomic activity* means that an activity works against the interests of another.

The veracity of this assertion could be proved if we were an external observer of all the systems and their environments in order to avoid any perturbation. This cannot exist in our physical world, so this is why it is defined as an axiom.

Lemma 1 *A cooperative system is functionally adequate.*

By definition, a cooperative system has only beneficial activities for its environment. So, there is no antinomic activity for the system and the previous axiom can be used.

Lemma 2 *For any functionally adequate system S there exists at least a cooperative system S* which is also functionally adequate in the same environment.*

The demonstration uses a thinking experiment in order to construct the system S* from the initial system S. It has four steps: specifying an algorithm to construct a cooperative system, showing the termination of the algorithm, proving that the new system realizes a function equivalent from the system S, and concluding that this is a right functionally adequate system S*.

Lemma 3 *Any system having an internal cooperative medium is functionally adequate.*

An internal medium of a system contains all its parts and physical supports needed for their exchanges. A system with cooperative internal medium has only cooperative exchanges with its environment because these exchanges are a subset of its parts interactions.

Lemma 4 *For any cooperative system, there exists at least a cooperative internal medium system that is also functionally adequate in the same environment.*

The method is identical to the lemma 2. The reasoning process involves all the system parts. The cardinality of the parts is assumed finite for any real system.

The theorem is easily obtained by operations of surjection and inclusion of the systems sets defined in the lemmas (functionally adequate, cooperative, cooperative internal medium).

3.3 The AMAS Technology

Our objective is to produce systems that perform well when they encounter unexpected difficulties. These difficulties are analogous to the “exceptions” in traditional programs. From an agent point of view, we call them Non Cooperative Situations (NCS). The designer has to describe not only what an agent has to do in order to achieve its goal but also which locally-detected situations must be avoided and, if they are detected, what to do about them (in the same manner that exceptions are treated in classical programs). More precisely three kinds of non cooperative situations should be detected by the agent:

- When an incoming signal is not understood or it is ambiguous;
- When new information does not cause any change or activity in an agent; and
- When the conclusions are not useful to others (i.e., they signal back their dissatisfaction or at variance with their expectations).

A cooperative agent in the AMAS framework has the four following characteristics. First, an agent is autonomous; that is an agent has the ability to decide to say “no” to an activity or start a new activity. Second, each agent is unaware of the global function of the system; that is the global function emerges (from the agent level towards the multiagent level). Third, an agent can detect non-cooperative situations and acts to return to a cooperative state. And finally, a cooperative agent is not altruistic in the sense that an altruistic agent is forced to help other agents. It is benevolent, i.e., it seeks to achieve its own goal while being cooperative, but not altruistic in the sense that it puts the goals of others above its own.

Generally, five features are necessary in order for a coherent collective behaviour to emerge from the interaction of the individual behaviours.

- Each agent has the skills necessary to be able to perform the partial function which is assigned to him.
- Each agent needs some knowledge of itself, the agents it interacts with and its local environment.

- Each agent is endowed with a social attitude which enables it to modify its interactions with other agents. This is based on what we call the cooperation: if an agent detects a non-cooperative situation, it acts to try and restore cooperation.
- An interaction language or protocol is necessary for the agents to communicate (whether the communication is direct or otherwise).
- Each agent has some “aptitudes” which are capacities to reason on its representations and its knowledge.

In this kind of adaptive MAS the designer has to give the agent the means to autonomously change its links with the other agents. We start from the idea that, to get good global behaviour the elements that constitute the system have to be “at the right place, at the right time” in the organisation. To achieve this, each agent is programmed to be in a cooperative situation with the other agents of the system. Only in this case, an agent always receives relevant information for it to compute its function, and it always transmits relevant information to others. The designer provides the agents with local criteria to discern between cooperative and non-cooperative situations. The detection and then elimination of non-cooperative situations between agents constitute the engine of self-organisation.

Thus the hope is that, depending on the real-time interactions the MAS has with its environment, the organisation between its agents emerges and constitutes an answer to the aforementioned difficulties (indeed, there is no global control of the system). In itself, the emergent organisation is an observable organisation that has not been completely specified by the designer of the system. Each agent computes a partial function, but the combination of all the partial functions produces the global emergent function. Depending on the interactions between themselves and the environment, the agents change their interactions, i.e., their links. This is the self-organisation in an AMAS system.

By design, the emerging purpose of a system is not recognizable by any part of the system. Rather all adaptation towards obtaining the desirable feedback is achieved in strictly local manner (relative to the activity of the parts which make it up).

4. Flood Forecast by Cooperative Self-Organizing Agents

In this part we exhibit a concrete application, called STAFF, demonstrating how improvements in the cooperation at a micro level (the system components) can imply a macro level improvement. The first subsection describes the overall architecture of the MAS. The second shows some typical result obtained with the system. The third presents the self-organisations rules between the

agents. And the last highlights the utility of cooperation by showing the results obtained when some cooperative rules are missing.

Part of the difficulty in forecasting floods results from the incapacity to exactly forecast the quantity and the location of rains. As long as the forecasting of rains is approximate, this environment will be effectively unpredictable. Thus, the design of STAFF uses a totally different approach from the existing models of forecasting (which rely on approximate physico-hydrological models). The aim of STAFF is to be able to compute a flood forecast at any point in the basin without any prior information (either physical or hydrological). It only utilizes the data gauges in the basin and the real current river level at the point for which STAFF must provide a flood forecast. It must give such forecasts in real time.

To achieve this the principles of autonomy, cooperation and self-organisation have been applied within the AMAS framework. We show here how the theory has been applied, setting out in detail the architecture and the working principles of the adaptive MAS. Since 2001, STAFF has been operational on the Sophie software platform (Georgé et al., 2003), and used by the experts from DIREN (Direction Régionale de l'Environnement) which are responsible of flood management in the region of France Midi-Pyrenees. The results presented here are obtained from real graphical results on this platform.

4.1 The STAFF Architecture

The upper level of STAFF is a MAS which computes a forecast for the water level as long as it is running. Each agent in this level (called "hourly agents") in effect, tries to give the change in water level which will occur during the next hour (for example between $t+3$ and $t+4$). Each hourly agent is a MAS itself, composed of agents on the second level. An agent in this second level is associated with a sensor it can be interpreted as trying to find the real influence of this measurement during that time period for the hourly agent.

In order to indicate the self-organisation steps, we will use the following notation:

- S_i is the raw value provided by the sensor agent associated with the sensor i (which should correspond indifferently to a rain gauge or a river gauge without any additional information on its location).
- ω_i is the weight which will be applied to the entry value i . This is an integer varying between 0 and 2000. A value less than zero means that this sensor is currently irrelevant to explain the output (the hourly forecast).
- $\Delta\omega$ is the minimal value of the increment applied to the weight ω_i . This value is 1 in the current system.

- The “forecast” provided by the hourly agent is \mathbf{F}_k (from time $t+k-1$ to time $t+k$) calculated as the weighted sum of all the entries greater than zero. $\mathbf{F}_k = \Sigma (\omega_i * \mathbf{S}_i)$, $\forall i/\omega_i > 0$. This is the change of the river level.
- The global forecast is the result given by the MAS station. It is the sum of the \mathbf{F}_k (for a forecast for time t four hours in the future, it is the sum of the changes during the four hours preceding t).
- The feedback to each station is the actual change in the river level between time $t-1$ and t .

4.2 Flood Forecast Results

In Figure 16.1, the X-axis represents time over 4 days of measures (96 hours). The Y-axis is the evolution of a river level (in meters). The dark-blue curve (thick line) is the real evolution of the river, whereas the red curve (thin line) is the forecast of the river given by STAFF. From evidence, we can say that the STAFF predictive model is very close to the real evolution of the river.

Classical physic-hydrological forecast models consist in generic formula, the parameters of which are determined from measures on ground and from historical data about floods in the relevant basin. Some of these are linear models that compute the forecast at $t+\Delta t$ for a station according to the flow growing between t and $t-\Delta t$ at the same station; others follow the principle of the “rain-flow” model, which is a function of a flood coefficient and the action delay of rain. Non-linear models are also used to simulate the attenuation of the flood wave between upstream and downstream stations. The building and adjustment of these formulas is time expensive (several months) and do not give good results for every point in the basin (particularly upstream stations and new stations) and for every kind of flood (particularly those that are slow or which have a constant evolution phase) or when the sensors break down.

Compared to these classical models, the STAFF system has many interesting properties that have been observed during its functioning:

- 1 Generally speaking, on the dozen locations where classical models exist in Midi-Pyrénées, the STAFF software gives results that are at least as good the previous ones and often better.
- 2 STAFF correctly predicts the initial period of the flood even when some of the input data is missing, by automatically compensating them with other entries which are not so relevant.
- 3 STAFF provides forecasts for river level change in real time that are useful for expert decision making, even where the input data are very noisy.

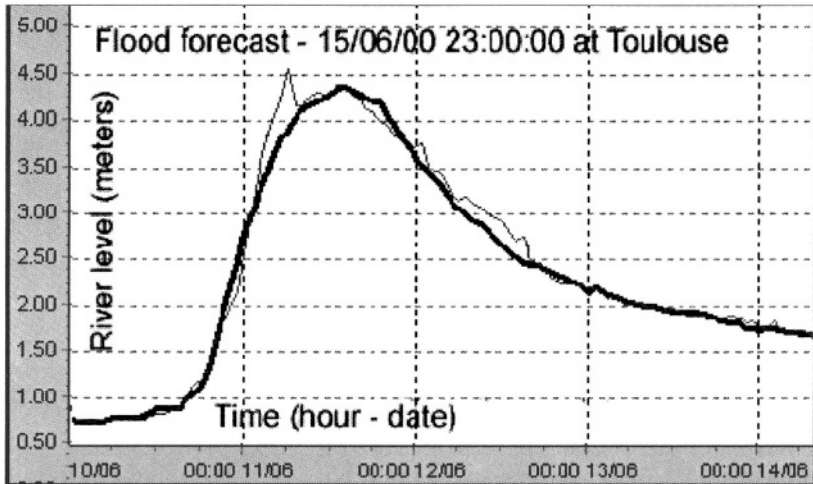


Figure 16.1. Typical functioning of STAFF

- 4 The system is able to provide flood forecasts for an upstream station by creating a model based on the rains moving from one basin slope to another, rather than a usual flow propagation (which is inaccurate in this case). Simply put, the system makes a forecast for one valley based upon the rainfalls in an adjacent valley because it “learned” that the clouds will probably move from that valley to the first.

These results firmly establish the effectiveness of the STAFF software and we can say that it is “functionally adequate” according to the terminology of the AMAS framework. In fact, the framework claims that when components of a system are in cooperative situation, the system has a suitable behaviour in the environment. As we will see, it is only by reducing conflicts between forecast agents that STAFF reduces highly the forecast error.

4.3 Self-Organisation in the MAS

To approximate the measure curve by a linear function the specific influence of each item of sensor information has to be determined. Due to the highly dynamic nature of a flood, this influence is unknown at the design phase of the system. This is the reason we encapsulate each sensor with an agent that is in charge of determining this influence as a weight. Each entry (typically one thousand of them) comes from sensors in the Garonne hydrological basin. At a station, all the sensors intervene as a pondered sum of their weights in order to compute the change in water level over one hour. Thanks to this systematic

adjustment, the modelling is adaptive and the relation non-linear over time in spite of using a simple balanced sum, since the weights will change over that time.

The aim of a sensor agent is to adjust the extent to which the measure it is associated with affects the hourly forecast. The typical non-cooperative situation for a sensor agent consists in a bad evaluation of its influence. This case appears every time the hourly agent it is working for has to readjust its forecast. Non-cooperative situations depend on the notions of correlation and influence. Correlation indicates whether a measure has to be used to compute the forecast and if so the level of influence. There are three types of non-cooperative situations a sensor agent may face:

- 1 The entry value of an agent is not correlated with the feedback (the station evolution). This is a non-cooperative situation of uselessness because the agent cannot, at this moment, explain the output. In this case the weight must be diminished:

$$\omega_i := \omega_i - \Delta\omega.$$

- 2 The entry value of an agent is correlated with the feedback, but ω_i is currently negative (this entry was in the past mainly uncorrelated). This agent could be useful and the weight value must be increased:

$$\omega_i := \omega_i + \Delta\omega.$$

- 3 The entry is correlated but the forecast given previously (i.e., the forecast for the current time t , for which we know now the feedback value) is erroneous. The wrong influence of the agent must be modified in order to decrease the forecast error. Because at the same time many agents can act in the same way we are in the concurrency situation and each agent has to do some small adjustment to act but by taking into account the actions of the other agents:

$$\omega_i := \omega_i + \text{Sign}(\text{Feedback} - \text{Forecast}) * \Delta\omega.$$

In order to take into account the very latest sensor information, the forecast delay for a hourly agent has to be the shortest possible. The inferior limit for the forecast delay is equal to the range period of the sensors which is one hour for the Garonne basin.

Thus we have built a MAS made of several forecasting agents (the hourly agents), each having a forecast range of one hour. The number of these agents in the MAS at the station depends on the number of hours in the forecast delay. Each such agent computes its forecast for its own period of one hour. The sum of the hourly agents gives the final forecast associated with the station.

The presence of several hourly agents in the same system may lead to conflict situations between their respective results. STAFF is programmed to take

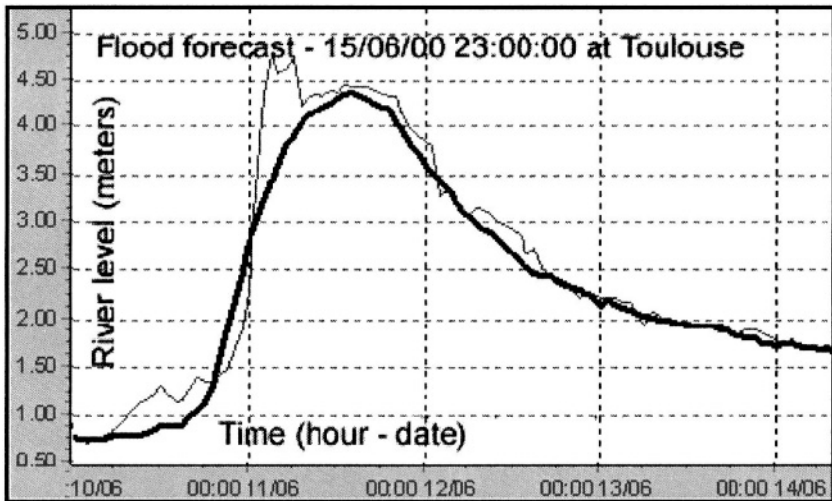


Figure 16.2. STAFF results without cooperation between the hourly agents

into consideration only the most critical of them leading to the following treatment:

- 1 The first forecasting agent F_1 compares its previous forecast with the last station measure (Feedback). A difference is a conflict between the forecast value and the real value, which implies an adjustment inside the corresponding hourly agents.
- 2 F_2 (giving the change forecast between 1 and 2 hours in the future) compares its previous forecast (the one it gave 1 hour in the past) with the new forecast of F_1 (so for the same time in fact). The difference is interpreted as a conflict that implies an adjustment inside the corresponding hourly MAS. This is repeated with the other forecast agents.

4.4 Results without Cooperation

In the following experiments indicated in Figure 16.2 we have suppressed the adjustment corresponding to the second point indicated in the previous paragraph. The results are given for exactly the same basin location and the same time as the results given in Figure 16.1 (the thick curves are identical).

The STAFF forecasts without this mechanism for enforcing coherence are worse than those with cooperation. One might think that this specific cooperation removal would not be very influential because it only corresponds to an adaptation process based on the difference of results given by the different

forecasting agents and not on what is usually called “feedback” in learning methods. However this is not the case. Indeed, despite the fact that all these results are individually erroneous, the process of trying to diminish their differences results in an improved collective result.

If we refer again to the Functional Adequacy Theorem, we see that it says that there exists a system having a cooperative internal medium which gives the right result to the environment. Here, STAFF’S environment is the user who wants a good forecast, and when this is not the case STAFF receives a feedback giving the real observed variation level of the water at the current time. This starts an adjustment process for the hourly agent F_1 (from time t to time $t+1$). Strictly speaking, this is the only learning that can be done given the feedback; the predictions of the other agents took place too long ago for them to take into account the current feedback. But the cooperative agent F_1 must also inform neighbours of its new potentially useful result. Particularly, the F_2 agent has done a forecast for the same moment but during the previous hour ($t-1$). If these two forecasts are different there is a conflict between F_1 and F_2 which must be solved in order to obtain a more cooperative internal medium for STAFF, according to the theorem. In fact, this cooperative mechanism is a way for all the agents to take into account the changes affecting the other agents because they think that these changes have some reason to exist (which is the case for F_1 since it received a feedback from the environment).

But the theorem gives no information about the parts (the agents) which are the constituents for the cooperative internal medium in a given application. So we have spent a lot of time during the development of STAFF to identify the F_k agents and we still are not sure that is the only relevant (or best) identification. The ADELFE methodology (see section 5.2 and chapter 8) gives some indications of how to do that, but this can be more or less difficult in a given application. This identification is an important and difficult part of the work but should always be possible because the theorem asserts that there exists at least one of this particular internal medium where all the parts are in cooperative interactions.

5. Software Engineering Requirements for Self-Organizing MAS

5.1 A Framework for Software Engineering

Software engineering concerns the establishment and use of sound principles (methods) in order to economically obtain software that is reliable and works on real machines (Bauer, 1972). Could we apply a software engineering method if the software function results from an emergence? The response will be positive if we can use emergence to guarantee a functional adequacy (i.e.,

reliable system). For this reason we must develop a systematic and disciplined approach that can deliver quantifiable benefits.

We are far from being able to definitely give such a positive answer. However it does appear that the approach of utilising agent adaptation using feedback (that is ultimately derived from the appropriateness of the global behaviour) and by concentrating on programming for adaptation of agent sub-components when it encounters unsatisfactory interaction in the system could allow for some optimism in this regard. The example in this chapter shows that something like it can be an effective way forward in, at least some, circumstances. The flexibility and homeostatic abilities of such systems does mean that they are a promising avenue to investigate.

Theory will play a part in this development. Some of the theory which demonstrates the theoretical adequacy of this approach was summarised above and, no doubt, more theory can be and will need to be developed. However the complexity of these kinds of systems and the nature of emergence will mean that *prior* theory will probably never be sufficient on its own (Edmonds, 2003b). Instead the development of relevant theory will have to be achieved more by methods akin to the natural sciences than those derived from the formal sciences. In other words, many of the theories and properties will have to be discovered and confirmed experimentally. However, as in the natural sciences, this does not mean that the theories will not be formal, just that they will be *contingent* theories whose reliability is established inductively and where it is unlikely there will be any final proof.

Thus it is likely that (as in biology) there will be many *kinds* of emergent system, each of which will need to be investigated and understood differently (Edmonds, 2003a). There is unlikely to be any substantial amount of generally applicable theory. Rather there may be a range of useful approaches and frameworks (like the AMAS framework described herein). What one can imagine is that, like architecture, there will be a range of well-understood and validated techniques that can be flexibly combined to construct workable systems, just as arches, beams, suspension etc. form part of a vocabulary of forms in constructing buildings.

5.2 The ADELFE Methodology for AMAS Systems

For a theory to be used to build artificial systems, even systems based on emergence, we need methodologies to concretely guide the engineers. In the particular case of the AMAS theory, a specific methodology called ADELFE (see chapter 8) is being developed and is the focus of chapter 8. ADELFE guides mainly the engineer by:

- Identifying the agents in the system even when there is more than one level of agents (an agent can be itself constituted by agents, thus enabling its adaptation); and
- Defining what are the Non Cooperative Situations an agent may encounter and the corrective actions he has to take.

5.3 Expected Benefits and Difficulties

In general, no mechanism for adaptation or learning which imposes the knowledge of a cost function is better than any other, whether the algorithm is distributed or centrally controlled. As the “*No Free Lunch Theorem For Search*” puts it “*In our investigation of the search problem ... the first question we addressed was whether it may be that some algorithm A performs better than B, on average. Our answer to this question, given by the NFL theorem is that this is impossible*” (Wolpert and MacReady 1995). The moral of this is that any algorithm of this kind needs to exploit some context-specific knowledge of the domain of application in order to gain any efficiency. In other words there is always a trade-off between the generality of an approach and its efficiency.

In the case of AMAS, as described here, the direct benefit is probably a greater generality of adequate performance as compared to non-adaptive MAS. That is, there is reason to suppose that AMAS systems will respond tolerably well to a greater range of conditions than non-adaptive versions. In the case above the STAFF system produced adequate forecasts in a highly dynamic situation. Outside this scope they will fail as much as non-adaptive MAS (and maybe even more catastrophically). Also it is likely that they will not reach the heights of efficiency (or accuracy) that may be possible for a highly-designed but specific solution. This is why emergent systems approaches are only relevant when usual design are inapplicable as quoted in the introduction.

From the point of view of the difficulty of producing a system of a suitable type AMAS shifts the burden. Using AMAS allows engineers to avoid specifying an explicit goal and allows them to focus on feedback and action under NCS. The increased flexibility of AMAS means that, if the internal system of co-adjustment is working alright, it will probably give an adequate (or near-adequate) result, and this means that if this level of performance is acceptable to the engineer then this makes the design and testing tasks somewhat easier. However the reliability of an AMAS will always be a contingent fact establishable by experiment and not establishable by formal means. When an AMAS system fails, the debugging will be probably difficult as there are many complex interactions and no reliable way of analysing the system to establish fault (in fact fault may be as distributed as the computation).

In summary AMAS probably has increased generality and flexibility but possibly at the cost of peak efficiency and theoretical certainty.

6. Conclusion

The work herein described is part of a wider current of work which seeks to understand complex emergent systems and, ultimately apply any useful discovered properties in the production of useful software systems. For this to be an acceptable way forward it needs to be established that systems that exploit “emergent computational processes” (in the widest sense) can be relied upon. Such reliance will never be amenable to formal demonstration, since there will almost certainly always be a finite (but hopefully very small) probability of failure. With emergent properties failure can never be completely ruled out.

However, in return for a lack of theoretical certainty, we hope that systems exploiting “emergent computational processes” can demonstrate a greater reliability *in practice* by being able to produce acceptable responses in unexpected situations via mechanisms of internal adaptation. If the global cost function was accessible to the agents on the lower levels, so that it could be utilised to describe, explain or predict the phenomena arising at the upper level (the system), then the system behaviour would not be emergent. Thus in emergent systems the agents at the lower level can not have (complete) access to the global cost function. This means that there will always be unknown aspects to their environment which can not be fully anticipated by them or a designer. For this reason the agents have to be, at least somewhat, adaptive so that they will be able to cope (at some level) with these unexpected aspects. Given the pervasiveness of situations with just such uncertainty about, we need some approaches to guide our partial specification of components so that the whole system behaves tolerably well. This chapter suggests that focussing upon the non-cooperative situations from the point of view of each agent is a productive way forward.

Theories about the scope and working of these computational mechanisms are to be sought for, so as to aid in the construction of reliable systems and guide further exploration. However these are not likely to be of the formal *a priori* kind common in many areas of computer science, but of a contingent nature established by the classic scientific experimental method.

Chapter 17

ENGINEERING SWARMING SYSTEMS

H. Van Dyke Parunak and Sven A. Brueckner

Abstract Most MAS are inspired by classical AI, whose objective was to realize human-level intelligence in a computer. As the field has moved toward multiple agents, there has been a presumption that individual agents still aspire to high-level intelligence. Swarming systems follow an alternative model, inspired more by artificial life than artificial intelligence. The individual agents in these systems may be non-cognitive, but complex, robust cognition emerges from their interactions. This chapter defines swarming and the concepts of self-organization and emergence that underlie it. It describes the kinds of problems for which it is well suited, explores why it functions, and outlines some initial principles of an engineering methodology for developing artificial swarming systems.

1. What is Swarming?

We define swarming as “useful self-organization of multiple entities through local interactions.” We begin by reviewing other definitions, then focus in on organization and self-organization, and the relation of these concepts with emergence.

1.1 Swarming

Definitions of swarming have been proposed by insect ethologists, roboticists, and military historians. Of the many definitions that have been proposed, a few will illustrate the main themes.

Students of biological systems use it to model decentralized self-organizing behavior in populations of (usually simple) animals, e.g., (Bonabeau, 2003; Bonabeau et al., 1999; Camazine et al., 2001; Parunak, 1997). Swarming has been defined, e.g., (Bonabeau et al., 1999), as “distributed problem-solving devices inspired by collective behavior of social insect colonies and other animal societies.” Table 17.1 lists a few examples that have been studied.

The use of the term to describe artificial systems can be traced to Beni, Hackwood, and Wang in the late 1980's (Benni, 1988; Benni and Hackwood,

Table 17.1. Some examples of swarming in nature

Swarming Behavior	Entities
Pattern Generation	Bacteria, Slime Mold
Path Formation	Ants
Nest Sorting	Ants
Cooperative Transport	Ants
Food Source Selection	Ants, Bees
Thermoregulation	Bees
Task Allocation	Wasps
Hive Construction	Bees, Wasps, Hornets, Termites
Synchronization	Fireflies
Feeding Aggregation	Bark Beetles
Web Construction	Spiders
Schooling	Fish
Flocking	Birds
Prey Surrounding	Wolves

1992; Benni and Wang, 1989; Benni and Wang, 1991; Hackwood and Beni, 1991; Hackwood and Beni, 1992). Their work focuses on populations of cellular robots, and they use the term to describe self-organization through local interactions. In the context of unpiloted air vehicles (UAV), Clough defines a swarm as a “collection of autonomous individuals relying on local sensing and reactive behaviors interacting such that a global behavior emerges from the interactions” (Clough, 2003). He distinguishes swarming (resulting from reactive behaviors of simple homogeneous entities performing simple tasks) from the emergent behavior of heterogeneous teams of deliberative entities performing complex tasks.

Recently, “swarming” has come into vogue in the military to describe a battlefield tactic that involves decentralized, pulsed attacks (Arquilla and Ronfeldt, 2000; Edwards, 2000; Edwards, 2003; Inbody, 2003). Military historians focus less on the process of self-organization and more on the resulting organization itself: “the systematic pulsing of force and/or fire by dispersed, internetted units, so as to strike the adversary from all directions simultaneously” (Arquilla and Ronfeldt, 2000); a “scheme of maneuver” consisting of “a convergent attack of several semi-autonomous (or autonomous) units on a target” (Edwards, 2003). The connection with insect applications is not coincidental. Insect self-organization is robust, adaptive, and persistent, as anyone

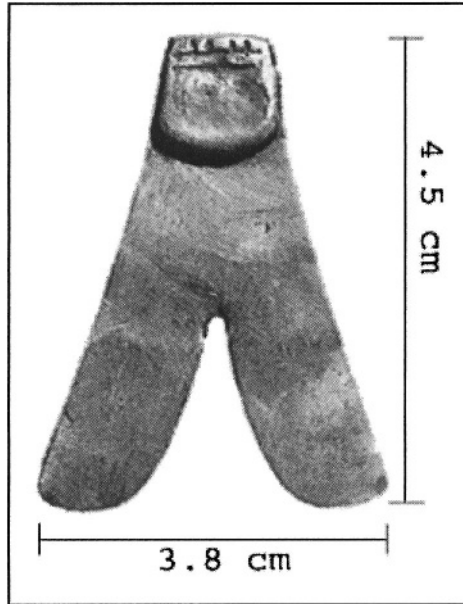


Figure 17.1. Egyptian “fly” medal for military heroes, 1550 BC (National Gallery of Art)

can attest who has tried to keep ants out of the kitchen or defeat a termite infestation, and military commanders would love to be able to inflict the frustration, discomfort, and demoralization that a swarm of bees can visit on their victims. The linkage between swarming and warfare is ancient. In the Bible, God promises to demoralize the indigenous population of Canaan before the invading Israelites in the words, “I will send the hornet before you” (Exodus 23:28; cf. Deuteronomy 7:20; Joshua 24:12). In the eighteenth dynasty (1550 BC), the ancient Egyptians awarded military heroes a gold and silver medal in the form of a stylized fly (Figure 17.1) (Hornung and Bryan, 2002), and there is evidence that the ancients sometimes hurled hives of stinging insects against their enemies (Neufeld, 1980).

For the purpose of this chapter, we will define swarming as “useful self-organization of multiple entities through local interactions.” This definition highlights elements of the others that have been suggested.

“Useful” emphasizes that we are interested in engineering systems that are answerable to someone outside of the system boundary for their behavior. Some forms of self-organized behavior, such as riots and oscillation, might be interesting to a biologist, but undesirable in a commercial or military application.

Self-organization is most prominent in the robotic definitions, since the concern there is to distinguish swarming from conventional top-down control schemes. The military definition does not emphasize self-organization, perhaps because of a historic tradition of top-down centralized control. We do not require that the self-organization result from reactive rather than deliberative individual behavior. Thus our definition includes not only Clough's "swarms" but also his "teams," if they meet the other terms of the definition.

The notion of *multiple entities* is common to all definitions, and indeed is intrinsic to the common-sense use of the term. A major motivator for swarming is the proliferation of autonomous platforms, such as vehicles, communications systems, and sensor systems. Although these systems are often referred to as "unmanned," in current practice it would be more accurate to describe them as "remotely manned." The flight crew for a Predator UAV consists of two people. Housing them in a control van rather than on board the flying platform considerably reduces their risk, but does not reduce the manpower requirements for fielding the vehicle. A major promise of swarming is multiplying the number of platforms that a single person can effectively control.

Our focus on *local interactions* has two motivations: a need and a promise. The need is a growing concern about communication congestion. The promise is the observation that local interactions suffice to maintain long-range coordination in biological systems, so that we ought to be able to reverse-engineer the underlying mechanisms for use in synthetic systems.

1.2 Organization

As used in expressions such as "self-organization," the word "organization" has at least three distinct, but related, meanings: it can refer to a mapping, a process, or a structure.

*Organization*₁ is a mapping from a system to an ordered set, e.g.,

$$Org : Systems \rightarrow \mathbf{R}^+$$

Such a mapping permits us to say that one system is "more organized" than another (or than the same system at a different time).

Different detailed definitions for this mapping are found in the literature. Common themes will include *entropy* and *symmetry*, as illustrated in Figure 17.2.

Denote a system by an upper-case letter, and its elements as the same letter in lower-case, indexed. Thus $A = \{a_1, \dots, a_n\}$ and $B = \{b_1, \dots, b_k\}$ denote two systems. The entropy of a system A is denoted by $S(A)$. With these concepts, we can meaningfully assert $Org(A) > Org(B)$ if

- $S(B) > S(A)$ or

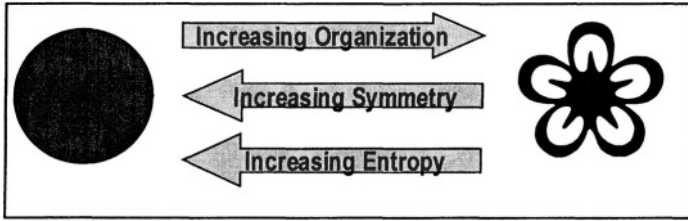


Figure 17.2. Symmetry vs. organization

- B has a higher order of symmetry than A .

Entropy can be computed against different bases, such as the spatial distribution of agents, their directions of movement, the behaviors open to them at any moment, or the time series generated by their actions. This variety can lead to the concern that entropy, and thus *Organization*₁, is in the eye of the beholder. In fact, methods exist for defining changes in entropy in an unambiguous way (Crutchfield, 1994; Shalizi, 2001), though discussing them in detail is beyond the scope of this survey.

*Organization*₂ is a *process* in a single system in which *Organization*₁ increases with time:

$$Org(A(t_2)) > Org(A(t_1)), t_2 > t_1$$

*Organization*₃ is the *structure* resulting from *Organization*₂, and can be measured with *Organization*₁.

1.3 Self-Organization and Emergence

With this understanding of “organization,” it would seem natural to define “self-organization” as a process (*Organization*₂) that reduces the entropy of a system without external intervention (motivating the modifier “self”). This definition is in line with some that have been proposed in the literature, for example:

- Camazine (Camazine et al., 2001): “Pattern formation occurs through interactions internal to the system, without intervention by external directing influences (leaders, blueprints, recipes, templates).”
- Bonabeau (Bonabeau et al., 1999): “A set of dynamical interactions whereby structures appear at the global level of a system from interactions among its lower-level components. ...The rules specifying the interactions are executed on the basis of purely local information, without reference to the global pattern.”

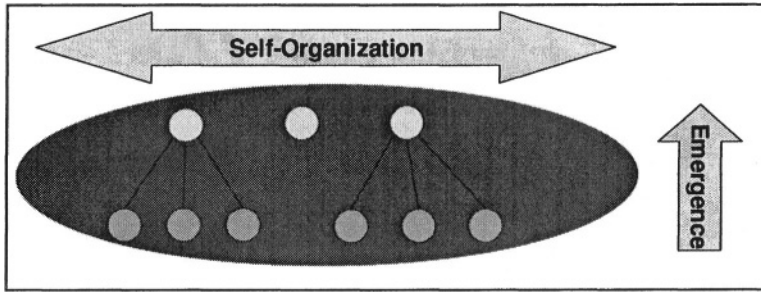


Figure 17.3. Comparing self-organization and emergence

These definitions emphasize the system boundary (through terms such as “local” and “internal”). The second includes three other concepts as well:

- A distinction of multiple levels within a system;
- “Interactions” among entities at lower levels of the system; and
- The “appearance” or “emergence” of properties and structures at higher levels from these interactions.

Other definitions of “self-organization” rely only on these three themes, without focusing on the system boundary, for example:

- Biebricher (Biebricher, C. K., 1995): The process by which individual subunits achieve, through their cooperative interactions, states characterized by new, emergent properties transcending the properties of their constitutive parts.
- Schweitzer (Schweitzer and Zimmermann, 2001): The emergence of new system properties not readily predicted from the basic equations.

To achieve greater precision, we propose distinguishing between emergence and self-organization on the basis of the contrast between the horizontal concept of system boundary and the vertical concept of levels (Figure 17.3).

We define *Self-Organization* as organization:

- Among elements *within a level*; and
- Without information flow across the boundary.

The second law of thermodynamics demands that there be some *energy flow* across the boundary of any system whose organization increases over time. *Self-organization* requires that this energy flow not contain information. This

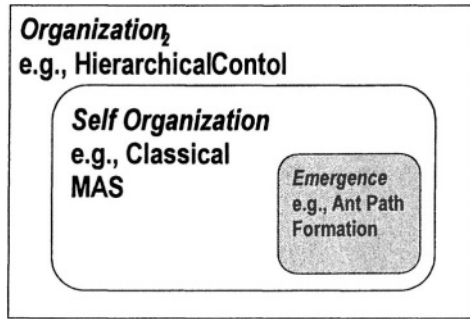


Figure 17.4. Emergence as a subcategory of self-organization

definition depends critically on the location of the *system boundary*. If the boundary is moved, a system's character as self-organizing or not may change.

We define *Emergence* as a subcategory of self-organization (Figure 17.4). Emergence (as we use the term) describes the appearance of structures at a *higher level* that are not explicitly represented in *lower-level* components. The reliance of swarming systems on locally available information makes it difficult for them to reason explicitly about higher-level structures, so emergence tends to be an important mechanism in swarming systems.

Neither self-organization nor emergence is necessarily good. The formation of structures will correspond to a reduction in entropy, whether those structures support or frustrate the objectives of the system stakeholders. The fact that emergent structures can be pathological (as in the case of race conditions or herding behavior) may explain the apprehension with which some people view emergence. For example, Wooldridge and Jennings assert (Wooldridge and Jennings, 1998), "Emergent functionality is akin to chaos ...". They urge engineers of agent systems to "severely restrict the way in which agents can interact with one-another ... ensure that there are few channels of communication between agents ... restrict the way in which agents interact" in order to reduce the likelihood of emergent behavior. A consequence of this restriction is that any desired system-level behavior must be explicitly represented in the lower-level components, a requirement that is difficult to meet if the system's requirements include responding gracefully to unanticipated changes in its environment. Our alternative approach is to develop principles for designing and developing systems whose emergent behavior is beneficial or at least benign.

This difference in vision leads to two distinct approaches to building MAS (Figure 17.5).

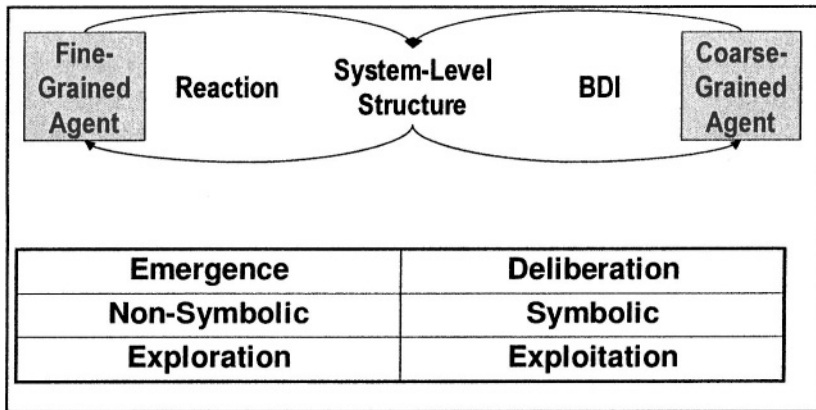


Figure 17.5. Two families of self-organizing systems

- Classical MAS achieve self-organization through deliberation among fairly sophisticated (“coarse-grained”) agents. Emergent systems can use much simpler reactive agents.
- Reasoning in emergent self-organization is often non-symbolic, while classical systems are usually symbolic.
- Because of the need for representing system-level behavior explicitly at all layers, non-emergent systems are best suited for exploiting well-known environments. The ability of emergent systems to produce new behaviors is appropriate for more exploratory problems.

A front line of our current research is understanding how to hybridize these two families of systems.

1.4 Alternatives to Self-Organization

(Camazine et al., 2001) identifies four alternatives to self-organization: leaders, recipes, blueprints, and templates.

- A *leader* is a single agent that receives status information from the other agents, decides on the action that each should take, and issues commands. This paradigm is sometimes called “centralized control.” If the leader is not part of the system, the flow of the leader’s commands to the other agents crosses the system boundary.
- A *recipe* is a script (a process description that is sequenced in time) that is constructed by the system’s designer and installed at compile-time.

If the designer is not part of the system, this script crosses the system boundary when it is installed.

- A *blueprint* is a map (a spatial prescription) that is constructed by the system's designer and installed at compile-time. If the designer is not part of the system, this map crosses the system boundary when it is installed.
- A *template* is a structure in the system's environment (e.g., the walls of a soccer arena) that constraints the system's behavior. If this structure is not part of the system, information about it crosses the system boundary as the system interacts with it.

It is useful to keep these alternatives in mind for two reasons. First, self-organization is not the best answer for every problem. In some cases, an alternative approach may be preferable, and responsible engineering requires awareness of these alternatives. Second, in each case, the system can be made self-organizing by expanding the system boundaries to include the source of the information. Thus many agent architectures include a (software) leader that directs the actions of the other agents. The engineer must carefully specify the system boundary, and may be able to adjust the behavior of the system significantly by shifting the boundary.

2. Where would You Want to Use Swarming?

Five domain features indicate the appropriateness of swarming: discreteness, deprivation, distribution, decentralization, and dynamism.

2.1 Discrete

It is easiest to apply agents (whether self-organizing or not) to a domain if the domain consists of discrete elements that can be mapped onto the agents. Some forms of organization also are achieved most naturally in a discrete system, for example, those that are characterized as a graph structure of some sort.

2.2 Deprived (Resource-Constrained)

We say that a system is “deprived” (or resource constrained) when limits on resources (such as processing power, communications bandwidth, or storage) rule out brute-force methods. For instance, if enough communications bandwidth is available, every agent can communicate directly with every other agent. If agents have enough processing power, they can reason about the massive input they will receive from other agents. If they have enough storage,

they can maintain arbitrarily large sets of instructions telling them what to do in each circumstance.

Under such assumptions, swarming architectures would seem to have little benefit. Some futurists extrapolate the historically exponential increases in hardware processing power, storage, and bandwidth, and claim that these constraints will quickly disappear. At the hardware level, Moore's law and its analogs for bandwidth and storage give good reason to be optimistic. However, a computer system is more than hardware. It is constrained by theoretical, psychological, commercial, and physical issues as well. For example:

- No matter how much storage is available, the knowledge engineering effort required to construct large knowledge bases remains a formidable *psychological* obstacle to completely defining the behavior of every single agent.
- No matter how fast processors get, the *theory* of NP-completeness points out that the time required to solve reasonably-sized problems in many important categories will still be longer than the age of the universe. An important instance of this challenge is the truth maintenance problem, the challenge of detecting inconsistencies in a knowledge base that result from changes in the world, which is NP-hard for reasonably expressive logics.
- No matter how much bandwidth the hardware can support, the *market* may not make it available in the configuration needed for a specific problem. Military planners, for instance, have long counted on the availability of commercial satellite channels, but the commercial market has moved toward land-based fiber backbones, resulting in a major shortfall in projected available bandwidth for military deployments in underdeveloped areas.
- The growing emphasis on Pervasive Computing and nanotechnology requires the deployment of computation on very small devices. The *physical* limitations of such devices will not permit them to support the level of processing, storage, and communications that can be realized on unconstrained devices.

Several characteristics of swarming systems make them good candidates for deprived environments. For example:

- Interactions among system components are typically local. If information needs to move long distances, it does so by propagation rather than direct transfer. Local interactions limit the number of neighbors about whom each agent must reason at a time, and enable the use of low-power transmissions that permit bandwidth to be reused every few kilometers.

- Because system-level behaviors do not need to be specified at the level of each element, the knowledge engineering and storage requirements are greatly reduced.
- Emergent systems commonly maintain information by continuously refreshing current information and letting obsolete information evaporate. This process guarantees that inconsistencies remove themselves within a specified time horizon, without the need for complex truth-maintenance procedures.

2.3 Distributed

The notion of “local interactions” is central to our definition of swarming. Keeping interactions local is a powerful strategy for dealing with deprived systems, but requires that the entities in the problem domain be distributed over some topology within which interactions can be localized.

The most common topology is a low-dimensional Euclidean manifold, or a graph that can be embedded in such a manifold. For example, insect stigmergy takes place on physical surfaces that, at least locally, are embedded in two-dimensional manifolds. Most engineered applications of swarming such as path planning (Parunak et al., 2002b), pattern recognition (Brueckner and Parunak, 2002), sensor network self-organization, and ant-colony optimization (Dorigo et al., 1996), follow this pattern. In these applications, locality can be defined in terms of a distance metric, and enforced by physical constraints on communications (e.g., a node’s neighbors are all the other nodes with whom it has radio contact).

More recent work, e.g., in telecommunications (Heusse et al., 1998), or in our laboratory, on semantic structures, successfully mediates agent interactions via scale-free small-world graphs. Such graphs have long-range shortcuts and so are typically not embeddable in low-dimensional manifolds. These shortcuts pose problems for classical definitions of distance, but locality of interaction can still be defined in terms of nearest-neighbor graph connectivity, and the empirical success of these latter efforts shows that this form of locality is sufficient to achieve coordination.

2.4 Decentralized

As a system characteristic, decentralization is orthogonal to distribution. In a centralized system, all transactions require the services of a single distinguished element. If the system is not distributed, the central point and the system are identical. If it is distributed, the central point is one of the elements, with which the others must communicate. A common extension of centralization in a distributed system is the hierarchy, in which the central element for

a small group of nodes joins with other nodes at its level in reporting to a yet higher central element, and so on until the top node is reached.

Swarming can be a poor choice for applications that require centralization. The restriction to local interactions means that communications between peripheral elements and the central element is an emergent behavior of the system, which may not meet the quality of service requirements or the need for detailed predictability that often lead to a requirement for central control. However, systems designers should be cautious about accepting a centralized architecture. Such architectures have at least three weaknesses.

- They are inherently resistant to increases in *scale*. As the system grows, the capacity of the central element must also grow. In decentralized approaches, new elements can be added without changing any of the existing elements.
- A frequent role of the central element is to mediate interactions among lower-level nodes (as in the mediator architecture, see <http://ksi.cpsc.ucalgary.ca/projects/Mediator>). This technique may actually lengthen the communication path between two nodes, leading to undesirable *delays* as messages travel up, then back down, the hierarchy.
- The central element and the communication paths leading to it are vulnerable to attack or failure, making the system less *robust* than a swarming system.

Centralized architectures often result more from tradition than from absolute system requirements, and a growing body of cases suggests that acceptable functionality can be achieved, with improved scalability, timeliness, and robustness, in a decentralized way. In addition, centralization is impossible in some cases (such as achieving coordination among a population of entities whose members are not known in advance and who do not all have access to a common element). Swarming techniques are a natural candidate for implementing decentralized architectures.

2.5 Dynamic

A system is dynamic if its requirements change during its lifetime. The emergent behavior that is characteristic of swarming is a powerful way for dealing with changed requirements. The system elements do not need to encode the system-level behavior explicitly, and so do not need to be modified when those requirements change. Three aspects of such change affect the need for emergence: scope, speed, and obscurity.

Scope characterizes the amount of change to which a system's requirements are susceptible. The less the scope of change, the more likely it is that the system as originally configured will deliver acceptable performance. The greater

the degree of change, the more value there is in the ability of the elements to reorganize to produce new emergent behaviors that were not active in the initial configuration.

Speed characterizes the rapidity of change, and affects the desirability of swarming by way of the distinction between centralized and decentralized architectures. If the system changes slowly, non-swarmling techniques that rely on centralized organizations can tolerate the time delays imposed by hierarchical communications. As the rate of change begins to outpace the communications time through the hierarchy, centralized organizations find themselves perpetually providing the answers to yesterday's problems, and unable to respond rapidly enough. A common response is to flatten the organization and empower lower-level nodes to act on local information, essentially moving toward a swarming architecture.

Obscurity reflects the degree to which later requirements can be anticipated by the original designer. Even if changes are rapid and wide in scope, if they follow along the lines anticipated by the designer, simple parameter adjustments in a non-emergent architecture may be able to cope with them. Swarming systems are much better at enabling a system to satisfy requirements that would be surprising to its original designer.

3. Why does Swarming Work?

Swarming is a discovery, not an invention. It is a naturally occurring phenomenon that we seek to imitate in engineered systems. Design principles for effective artificial swarming systems must be developed from an understanding of why swarming works in natural systems.

We analyze these underlying principles of swarming in terms of three restrictions on the space of all possible multi-process systems, outlined in Figure 17.3.

- The various processes must be *coupled* with one another so that they can interact.
- This interaction must be self-sustaining, or *autocatalytic*. Autocatalysis enables self-organization, but it is not necessarily useful.
- The self-organizing system must produce *functions* that are useful to the system's stakeholders.

In discussing each of these, we first review the concept and its mechanisms, then discuss design principles to which it leads.

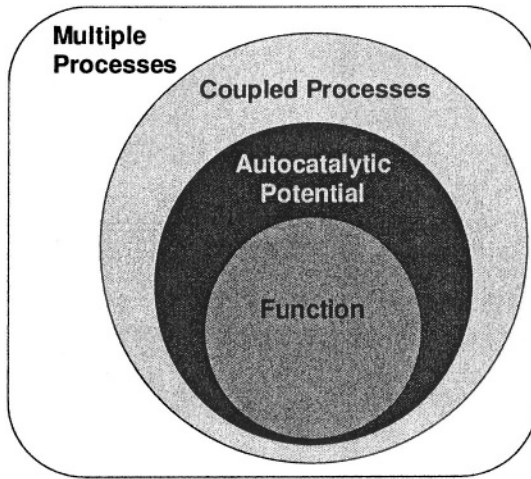


Figure 17.6. Three enablers for swarming systems

Table 17.2. Categories of Information Exchange

		Topology of Inter-Agent Relationships	
		Centralized (between Distinguished and Subordinate agents)	Decentralized (among Peer agents)
Information Flow	Direct (messages between agents)	Construction (Build-Time) Command (Run-time)	Conversation
	Indirect (non-message interaction)	Constraint	Stigmery (generic) Competition (limited resources)

3.1 Coupled Processes

How Processes can be Coupled. Agents must exchange information if they are to self-organize. Different patterns of information exchange are possible, and can be classified along two dimensions: Topology and Information Flow (Table 17.2).

The Topology dimension depends on two different kinds of relations that agents can have with one another. When the agents can say “No” to one another within the rules of the system, they are “peer agents.” When one of them (say agent A) can say “No” to the other (B), but B cannot say “No” to A, we

call A the “distinguished agent” and B the “subordinate.” The relationship between two agents may be fairly fixed (for example, the relationship between a human programmer and her software agent). Or it may vary over time (as when peer agents negotiate a work plan that calls for one of them to supervise the other, resulting in a distinguished-subordinate relationship during execution). These concepts can be developed more formally through dependency and autonomy theory (Castelfranchi, 2000; Parunak, 1990). Centralized information exchange is between a distinguished and a subordinate agent, while decentralized information exchange is between peer agents.

The Information Flow dimension relies on environmental state variables that the agents can manipulate and sense. All information exchange is ultimately mediated by the environment, but the role of the environment is sometimes not modeled explicitly. The information flow from one agent to another is Direct if no intermediate manipulation of information is modeled, and Indirect if it is explicitly modeled.

Centralized Mechanisms. They all involve communication between the distinguished agent and its subordinates. This flow may be direct (when the distinguished agent *constructs* or *commands* its subordinates) or indirect (when the distinguished agent constrains the subordinates by manipulating exogenous environmental variables visible to the subordinates). In correlation through command, used commonly in robot soccer, holonic manufacturing, and some simulation applications, agents behave much like objects, executing methods invoked by incoming messages. The focal point algorithm advocated by (Fenster et al., 1995) and the common utility functions implicit in (Genesereth et al., 1986) both rely on construction (common programming). In indirect centralized mechanisms, subordinates jointly sense changes in a shared exogenous environmental variable. The variable’s dynamics are independent of agent actions, so it cannot move information between subordinates. But it may serve as a synchronizing signal that correlates the agents’ actions. The experimenter who configures targets and obstacles in an experimental testbed is constraining the subordinates, supporting correlation through indirect centralized action.

Decentralized mechanisms all involve communication among peers. Most negotiation research focuses on direct peer-to-peer information flows (“conversation”). Indirect decentralized flows occur when peers make and sense changes to environmental variables. This class of coordination is called “stigmergy” (Grassé, 1959), from the Greek words *stigma* “sign” and *ergon* “work”: the work performed by agents in the environment guides their later actions. The information stored in the environment forms a field that supports agent coordination, leading to the term “co[ordination]-field” for this class of technique (Mamei et al., 2003b). Such techniques are common in biological distributed decentralized systems such as insect colonies (Parunak, 1997). A com-

Table 17.3. Varieties of stigmergy

	Marker-Based (Artificial signs inserted in the domain)	Sematectonic (Domain elements only)
Quantitative (Scalar quantities)	Gradient following in a single pheromone field	Ant cemetery clustering
Qualitative (Symbolic distinctions))	Decisions based on combinations of pheromones	Wasp nest construction

mon form of stigmergy is resource *competition*, which occurs when agents seek access to limited resources. For example, if one agent consumes part of a shared resource, other agents accessing that resource will observe its reduced availability, and may modify their behavior accordingly. Even less directly, if one agent increases its use of resource A, thereby increasing its maintenance requirements, the loading on maintenance resource B may increase, decreasing its availability to other agents who would like to access B directly. In the latter case, environmental processes contribute to the dynamics of the state variables involved.

Different varieties of stigmergy can be distinguished. One distinction concerns whether the signs consist of special markers that agents deposit in the environment (“marker-based stigmergy”) or whether agents base their actions on the current state of the solution (“sematectonic stigmergy”). Another distinction focuses on whether the environmental signals are a single scalar quantity, analogous to a potential field (“quantitative stigmergy”) or whether they form a set of discrete options (“qualitative stigmergy”). As shown in Table 17.3, the two distinctions are orthogonal.

Stigmergic mechanisms have a number of attractive features, particularly for swarming systems.

Simplicity. The logic for individual agents is much simpler than for an individually intelligent agent. This simplicity has three collateral benefits.

- The agents are easier to program and prove correct at the level of individual behavior.
- They can run on extremely small platforms, such as microchip-based “smart dust” (Pister, 2001).

- They can be trained with genetic algorithms or particle-swarm methods rather than requiring detailed knowledge engineering.

Scalable. Stigmergic mechanisms scale well to large numbers of entities. In fact, unlike many intelligent agent approaches, stigmergy *requires* multiple entities to function, and performance typically improves as the number of entities increases. Stigmergy facilitates scalability because the environment imposes locality on agent interactions. Agents interact with the environment only in their immediate vicinity. Increases in the number of agents are typically associated with an extension of the environment. The density of agents over the environment, and thus the processing load on each agent, usually does not increase.

Robustness. Because stigmergic deployments favor large numbers of entities that are continuously organizing themselves, the system's performance is robust against the loss of a few individuals. Such losses can be tolerated economically because each individual is simple and inexpensive.

Environmental Integration. Explicit use of the environment in agent interactions means that environmental dynamics are directly integrated into the system's control, and in fact can enhance system performance. A system's level of organization is inversely related to its symmetry (Figure 17.2), and a critical function in achieving self-organization in any system made up of large numbers of similar elements is breaking the natural symmetries among them (Ball, 1996). Environmental noise is usually a threat to conventional control strategies, but stigmergic systems exploit it as a natural way to break symmetries among the entities and enable them to self-organize. We make extensive use of stigmergy in our applications, building on the theoretical foundation and pheromone infrastructure outlined in (Brueckner, 2000).

The following are the design principles derived from coupled processes.

Coupling 1: Use a distributed environment. Stigmergy is most beneficial when agents can be localized in the environment with which they interact by sensing and acting. A distributed environment enhances this localization, permitting individual agents to be simpler (because their attention span can be more local) and enhancing scalability.

Coupling 2: Use an active environment. If the environment supports its own processes, it can contribute to overall system operation. For example, evaporation of pheromones in the ants' environment is a primitive form of truth maintenance, removing obsolete information without requiring attention by the agents who use that information.

Coupling 3: Keep agents small. Agents should be small in comparison with the overall system, to support locality of interaction. This criterion is not sufficient to guarantee locality of interaction, but it is a necessary condition. The fewer agents there are, the more functionality each of them has to provide, and the more of the problem space it has to cover.

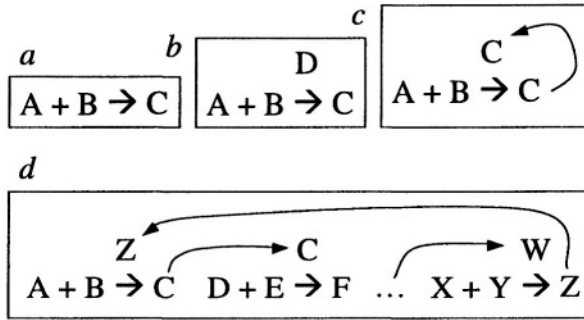


Figure 17.7. Relations among Processes, (i) A simple reaction; (ii) D catalyzes the conversion of A and B to C; (iii) An autocatalytic reaction; and (iv) An autocatalytic set of processes (shown as a ring, but other topologies are possible)

Coupling 4: Map agents to Entities, not Functions. Choosing to represent domain entities rather than functions as agents takes advantage of the fact that in our universe, entities are bounded in space and thus have intrinsic locality. Functions tend to be defined globally, and making an agent responsible for a function is likely to lead to many non-local interactions. For example, in a factory, each machine (an entity) has fairly local interactions with other machines, parts, and workers in its area of the plant, but a function (such as scheduling) must take into account all of the machines in the entire plant.

3.2 Autocatalytic Potential

What is Autocatalysis? The concept of autocatalysis comes from chemistry. A catalyst is a substance that facilitates a chemical reaction without being permanently changed. In autocatalysis, a product of a reaction serves as a catalyst for that same reaction. An autocatalytic set is a set of reactions that are not individually autocatalytic, but whose products catalyze one another. The result is that the behaviors of the reactions in the set are correlated with one another. If reaction A speeds up (say, due to an increased supply of its reagents), so does any reaction catalyzed by the products of A. If A slows down, so do its autocatalytic partners. This correlation causes a decrease in the entropy of the overall set, as measured over the reaction rates, so we would describe such a system as self-organizing. Figure 17.7 summarizes these concepts using reaction schemata.

Not all processes that are coupled, are autocatalytic. Autocatalyticity requires a continuous closed flow of information among the processes to keep the system moving. If the product of process A catalyzes process B, but process B's products have no effect (either directly or indirectly) on process A,

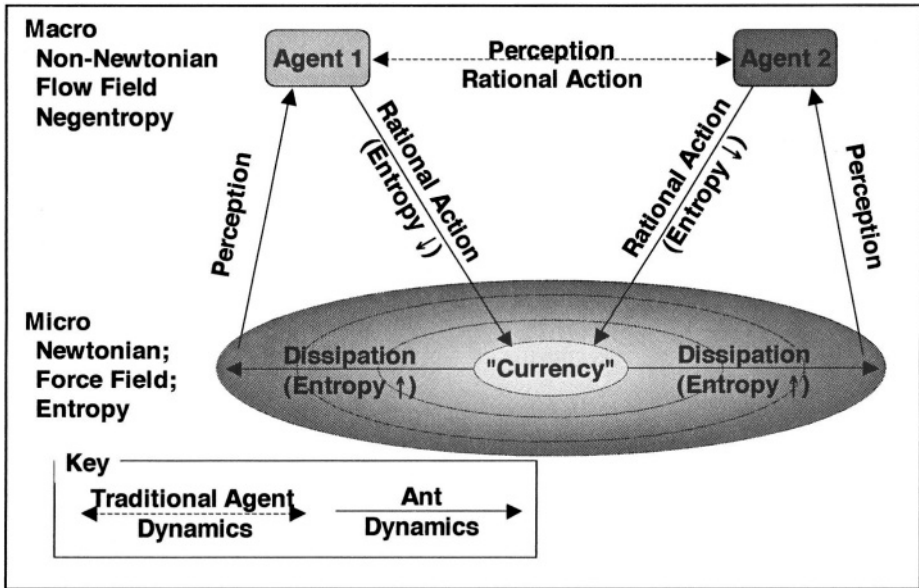


Figure 17.8. Autocatalytic flows in pheromone dynamics

the system is not autocatalytic. Furthermore, a system might be autocatalytic in some regions of its state space but not in others.

It is natural to extend this concept from chemistry to any system of interacting processes, such as a MAS. A set of agents has *autocatalytic potential* if in some regions of their joint state space, their interaction causes system entropy to decrease (and thus leads to increased organization). In that region of state space, they are *autocatalytic*.

Figure 17.8 exhibits two different approaches to achieving the closed information flow that supports autocatalysis in MAS.

- The dashed line between the two agents represents conventional agent interactions: the agents perceive one another, reason about how to coordinate their activities, and then act. The information flow in this case is maintained by the agents' perception of one another. As the number of agents increases, this approach requires an increasing amount of processing power on the part of each agent. If an agent is unable to sense or be sensed by other agents in the system, the information flow is broken, and the system's ability to self-organize will be reduced.
- The stigmergic approach used in swarming is represented by the solid lines forming a triangle of "Rational Action," "Dissipation," and "Per-

ception.” Agents deposit a “currency” (pheromone in the case of ants; money in the case of a market) into a shared environment. The aggregation of this currency from multiple deposits, and dissipative forces in the environment (evaporation in the case of pheromones), generate gradients that each agent can sense and to which it can respond. This continuous cycle provides the information flow that keeps the processes coordinated. Because the environment integrates the information from each agent, and because agents need sense only their local environment, the number of agents can increase without requiring more processing power on the part of each agent.

Two points are important to understand about autocatalyticity.

- In spite of the reduction of entropy, autocatalyticity does not violate the Second Law of Thermodynamics. The rationalization is most clearly understood in the stigmergic case (Figure 17.8). Entropy reduction occurs at the macro level (the individual agents), but the dissipation of pheromone at the micro level generates more than enough entropy to compensate. This entropy balance can actually be measured experimentally (Parunak and Brueckner, 2001).
- Information flows are necessary to support self-organization, but they are not sufficient. A set of coupled processes may have a very large space of potential operating parameters, and may achieve autocatalyticity only in a small region of this space. Nevertheless, if a system does not have closed information flows, it will not be able to maintain self-organization.

The following are design principles derived from autocatalysis.

Autocatalysis 1: Think Flows rather than Transitions. Our training as computer scientists leads us to conceive of processes in terms of discrete state transitions, but the role of autocatalysis in supporting self-organization urges us to pay attention to the flows of information among them, and to ensure that these flows include closed loops.

Autocatalysis 2: Boost and Bound. Keeping flows moving requires some mechanism for reinforcing overall system activity. Keeping flows from exploding requires some mechanism for restricting them. These mechanisms may be traditional positive and negative feedback loops, in which activity at one epoch facilitates or restrains activity at a successive one. Or they may be less adaptive mechanisms such as mechanisms for continually generating new agents and for terminating those that have run for a specified period (“programmed agent death”).

Autocatalysis 3: Diversify agents to keep flows going. Just as heat will not flow between two bodies of equal temperature, and water will not flow between

two areas of equal elevation, information will not flow between two identical agents. They can send messages back and forth, but these messages carry no information that is new to the receiving agent, and so cannot change its state or its subsequent behavior. Maintaining autocatalytic flows requires diversity among the agent population. This diversity can be achieved in several ways. Each agent's *location in the environment* may be enough to distinguish it from other agents and support flows, but if agents have the same movement rules and are launched at a single point, they will not spread out. If agents have different experiences, *learning* may enable them to diversify, but again, reuse of underlying code will often lead to stereotyped behavior. In general, we find it useful to incorporate a *stochastic element* in agent decision-making. In this way, the decisions and behaviors of agents with identical code will diversify over time, breaking the symmetry among them and enabling information flows that can sustain self-organization.

3.3 Function

How Systems Adjust to Required Function. Process interaction must support autocatalysis if a system is to support ongoing self-organization. From an engineering perspective, a further step is necessary. Self-organization in itself is not necessarily useful. Autocatalysis might sustain undesirable oscillations or thrashing in a system, or keep it locked in some pathological behavior. We want to construct systems that not only organize themselves, but that yield structures that solve some problem we need to address. There are two broad approaches to this problem, broadly corresponding to the distinction in classical AI between the scruffy and the neat approaches. It is likely that as the use of self-organizing systems matures, a hybrid of both approaches will prove necessary.

One approach, exemplified in amorphous computing (Abelson et al., 2000) and chapter 15, is to build up, by trial and error, a set of programming metaphors and techniques that can then be used as building blocks to assemble useful systems.

An alternative approach is to seek an algorithm that, given a high-level specification for a system, can compute the local behaviors needed to generate this global behavior. State-of-the-art algorithms of this sort are based not on *design*, but on *selection*. Selection in turn requires a system with a wide range of behavioral potential, and a way to exert pressure to select from this wide range of behaviors the ones that are actually desired.

One way to ensure a broad range of behavioral potential is to construct non-linear systems that can exhibit formally chaotic behavior. From a classical engineering perspective, chaos is undesirable because it is unpredictable in the long range. However, from an emergent perspective, chaos is desirable be-

cause it offers a simple way to sample a broad subset of the system's space of possible behaviors.

We can illustrate this somewhat nonintuitive insight with the simple logistic equation

$$x_{t+1} = gx_t(1 - x_t)$$

for $0 \leq x \leq 1$ and $1 \leq g \leq 4$. Figure 17.9 shows a plot of the 201st through 500th iterates of this function, starting at $x = 0.5$, for various values of g . The plot has three distinct regions.

- For $g < 3$, x converges to a single value, which depends on g . In this region, the system has only a single behavior for each value of g . If x is perturbed away from this value, it will quickly return. The system has no behavioral diversity.
- For $3 \leq g < 3.569945672 \dots$, x oscillates among a number of discrete alternatives. The Figure clearly shows regions with two, then four alternatives. In fact, as g approaches the upper limit of this range, the number of alternatives doubles repeatedly, so that a value of g can be found to yield any number of alternatives that is an integer power of two.
- For $3.569945672 \dots \leq g$, the system is formally chaotic. In this region, x varies widely over its range, and (left to its own behavior) never repeats its position exactly. This region offers the broadest behavioral potential for x .

The behavioral diversity evident in the chaotic regime is useful only if some way can be found to lock the system down to a particular behavior, but the basic mechanisms for such control have been known for over a decade (Ott et al., 1990). The basic idea is to let the chaotic dynamics explore the state space, and when the system reaches a desirable region, to apply a small control force to keep the system there.

It may seem that chaos is a complicated way to generate potential behaviors, and that it would be simpler to use a random number generator. In fact, virtually all such generators are in fact nonlinear systems executing in their chaotic regime.

In a MAS, the key to applying this generate-and-test insight is finding a way to exert selective pressure to keep the system balanced at the desired location. Natural systems have inspired two broad classes of algorithm for this purpose: synthetic evolution, and particle swarm optimization.

Synthetic evolution is modeled on biological evolution. Many different algorithms have been developed (Jacob, 2001), but they share the idea of a population of potential solutions that varies over time, with fitter solutions persisting and less fit ones being discarded. The variational mechanisms (which usually

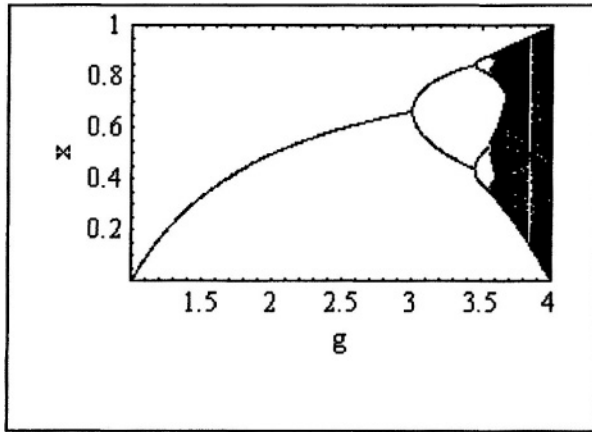


Figure 17.9. Behavioral diversity in the logistic function

include random mutation) explore the system's potential behaviors, while the death of less fit solutions and the perpetuation of more fit ones is the control pressure that selects the desired behavior.

Particle swarm optimization (Kennedy et al., 2001) is inspired by the flocking behavior of birds. In adaptations of this behavior to computation, solutions do not undergo birth and death (as in evolutionary mechanisms). Instead, they are distributed in some space (which may be the problem space, or an arbitrary structure), and share with their nearest neighbors the best solutions they have found so far. Each solution entity then adjusts its own behavior to take into account a blend of its own experience and that of its neighbors.

Market-based bidding mechanisms may be considered a variation on particle swarm optimization. The similarity lies in selection via behavioral modification through the exchange of information rather than changes in the composition of the population. The approaches differ in the use that is made of the shared information. In the particle swarm, agents imitate one another based on the information they receive, while in bidding schemes, they use this information in more complicated computations to determine their behavior.

The following are design principles derived from functional adjustment.

Function 1: Generate behavioral diversity. Structure agents to ensure that their collective behavior will explore the behavioral space as widely as possible. One formula for this objective has three parts.

- Let each agent support multiple functions.
- Let each function require multiple agents.

- Break the symmetry among the agents with random or chaotic mechanisms.

The first two points ensure that system functionality emerges from agent interactions, and that any given functionality can be composed in multiple ways. The third ensures a diversity of outcomes, depending on which agents join together to provide a given function at a particular time.

Function 2: Give agents access to a fitness measure. Agents need to make local decisions that foster global goals, an insight that is supported by formal analysis in Wolpert's Collective Intelligence (COIN) research (see <http://ic.arc.nasa.gov/projects/COIN>). A major challenge is finding measures that can be evaluated by agents on the basis of local information, but that will correlate with overall system state. Determining such measures is a matter for experimentation, although thermodynamic concepts relating short-range interactions to long-term correlations have the potential to yield a theoretical foundation. In one application, we have found the entropy computed over the set of behavioral options open to an agent to be a useful measure of the degree of overall system convergence (Brueckner and Parunak, 2003) that agents can use to make intelligent decisions about bidding in resource allocation problems.

Function 3: Provide a mechanism for selecting among alternative behaviors. If an adequate local fitness metric can be found, it may suffice to guide the behavior of individual agents. Otherwise, agents should compare their behavior with one another, either to vary the *composition* of the overall population (as in synthetic evolution) or to enable individual agents to vary their *behavior* (as in particle swarm optimization).

4. How can We Apply these Principles in Engineered Systems?

To illustrate the use of these principles, we briefly review several systems, described in more detail in other publications, that produce high-level cognitive behavior from swarming. In each case we review the problem being solved, summarize the behavior of the local elements, and discuss how they reflect the ten design principles outlined in section 3.

4.1 Pattern Recognition in a Sensor Network (Brueckner and Parunak, 2002)

The Problem. Driven by the need for greater efficiency and agility in business and public transactions, more and more data is becoming digitally available in real time on computer networks. These heterogeneous data streams reflect many aspects of the behavior of groups of individuals in a population (e.g., traffic flow, shopping and leisure activities, healthcare needs). A new

generation of active surveillance systems that integrate a large number of spatially distributed heterogeneous data streams may be used in various applications, for instance, to protect a civilian population from bioterrorist attacks, to support real-time traffic coordination systems, to trace collaboration structures in terrorist networks, or to manage public healthcare efficiently.

Active surveillance of population-level activities includes the detection and classification of spatio-temporal patterns across a large number of real-time data streams. Approaches that analyze data in a central computing facility tend to be overwhelmed with the amount of data that needs to be transferred and processed in a timely fashion. Also, centralized processing raises proprietary and privacy concerns that may make many data sources inaccessible. Our architecture avoids these problems through decentralization. Instead of transferring the data to a centralized processing facility, we transfer the processes (fine-grained agents) to the data sources. This architecture addresses both of these concerns. Access restrictions may be guaranteed through proven local processes. Bandwidth is reduced because long-distance communication of data is needed only when the network detects a pattern and needs to invoke a higher authority for action. Ultimately, one would like the response itself to be a distributed emergent response, but political realities suggest that in the immediate future self-organizing recognition systems will be much more acceptable than self-organizing systems that take action on people and property.

Summary of Architecture. We consider a distributed swarming agent architecture the most appropriate answer to the challenge of detecting spatio-temporal patterns in a network of heterogeneous sources of potentially proprietary real-time data. Instead of attempting to stream a tremendous amount of data into a central processing facility, we integrate the external sources into a network for mobile agent computing. Essentially, this network of agent processing nodes is a massively parallel computer for pattern detection and classification with a unique way of self-organizing the processing tasks.

Into our network of processing nodes we deploy large populations of simple mobile agents that coordinate their activities using stigmergy. Each node generates agents at a constant rate, and agents die after a fixed lifetime, thus ensuring coverage of the entire area under surveillance. Using artificial pheromones, the agents dynamically organize themselves around patterns observed in the data streams. The emergence of globally coordinated behavior through stigmergic interactions among many fine-grained software agents in a shared computational environment is facilitated by a component of the distributed runtime environment that emulates actual pheromone dynamics (aggregation, evaporation, dispersion) in the physical world. Our heterogeneous agent system continuously executes two parallel processes: pattern detection and pattern classification. More populations of agents could be deployed at any time, for

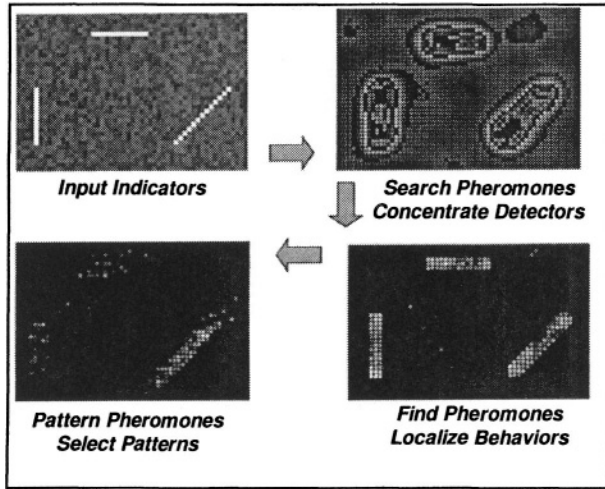


Figure 17.10. Stigmergic pattern detection and classification

instance to introduce additional criteria in the detection process, or to add more classification schemes.

The agents executing the detection process (“Detectors”) continuously process the input data and search for spatio-temporal structures, using two sets of flavors of pheromones. Detectors use Search pheromones to mark suspicious areas of the network and attract other detectors to confirm their discovery. A second set of Find pheromones, which require more deposits to stabilize, is used to record this confirmation, informing a local node that it is likely to be an instance of the pattern in question and enabling it to take appropriate action. Detectors search for unusually high differences in the data streams of neighboring locations in the network.

“Classifier” agents are responsible for the classification of the detected patterns according to a particular classification scheme. The pattern classification scheme used in our demonstration correlates the detected patterns with a particular, dynamically changing geographic direction (wind, modeling the dispersion of a bioterrorist weapon). The Classifiers move in a way that models the pattern being sought, and deposit a Pattern pheromone when they encounter a pattern that matches their behavior. Figure 17.10 shows the performance of the algorithm. The upper-left display is a grid in which each cell is set either to a random mixture of Red-Green-Blue, or to white. Viewing the overall display, we can see that the white cells are different from the mass of the other cells, and that they are arranged in extended patterns. However, a single cell with only local knowledge of its neighborhood can know neither of these facts. The upper-right display shows the Search pheromones deposited by Detectors

searching for unusual cells, based on their recent experience. The high propagation of these pheromones creates gradients that attract other Detectors for confirmation. As more and more Detectors agree that the cells are indeed unusual, Find pheromone (lower right) accumulates to mark the location of the unusual cells. Finally, “Classifier” agents moving diagonally across the field sense repeated Find pheromones aligned with their movement and mark them with Pattern pheromone to indicate an instance of a particular structure of interest.

Coupling 1: Use a distributed environment. The network of data collection nodes is distributed over space. For spatio-temporal pattern recognition, each data collection node maintains a temporal data structure to distribute agent interactions in time as well.

Coupling 2: Use an active environment. The environment implements the basic pheromone dynamics of Aggregation (fusion of observations from multiple agents), Propagation (communication), and Evaporation (truth maintenance).

Coupling 3: Keep agents small. Both data nodes and mobile agents are small compared with the overall system, and all interactions are local. No single agent can solve the problem. No data node can know on its own that it is part of a pattern being sought, nor can any individual Detector or Classifier confirm the detection or classification without collaboration by its colleagues.

Coupling 4: Map agents to Entities, not Functions. The data nodes correspond to distributed data sources in the physical domain. The Detectors and Classifiers are not domain entities, but neither does any one of them implement a function by itself. Detection and Classification emerge from the interactions of multiple Detectors and Classifiers. It is perhaps best to think of the Detectors and Classifiers as instances of hypotheses about structures in the environment, hypotheses that are confirmed or discredited through the stigmergic interactions.

Autocatalysis 1: Think Flows rather than Transitions. The fundamental information flow in this application is the pheromone loop illustrated in Figure 17.8.

Autocatalysis 2: Boost and Bound. Search pheromone exploits positive feedback: the more is deposited, the more Detectors come to that area and the more they deposit Search pheromone. If unrestrained, this reinforcement could lead to all Detectors becoming concentrated in one area, leaving other regions unexplored. Bounds on system dynamics are provided by the programmed death of agents and their continual rebirth at nodes distributed throughout the area.

Autocatalysis 3: Diversify agents to keep flows going. This architecture has three main species of agents among which information flows: data nodes, Detectors, and Classifiers.

Function 1: Generate behavioral diversity. Each function (detection and classification) requires multiple agents. It is less clear in this case that each agent performs multiple functions. However, agents do differ from one another.

- The birth location of each Detector or Classifier varies across the search area.
- A key behavioral parameter of mobile agents in this application is a threshold that indicates how distinct a data node must be from others that the agent has seen recently before it will deposit a pheromone. This threshold is randomly generated.
- Agents' movements, while influenced by local pheromone gradients, always incorporate a stochastic component. The pheromone strength in nearby nodes is used to weight a roulette wheel that determines the probability that the agent will move to each of those nodes in the next step.

Function 2: Give agents access to a fitness measure. The pheromone fields accumulate information about outlying nodes and extended patterns that combine the observations of many mobile agents that have followed different individual trajectories. Thus they are locally accessible repositories of information gathered over a much broader area, providing a local view of the global state of the problem.

Function 3: Provide a mechanism for selecting among alternative behaviors. Mobile agents adjust their detection thresholds using a variation of particle swarm optimization.

4.2 Searching and Imaging with Unmanned Air Vehicles (Parunak and Brueckner, 2003)

The Problem. Some sensing problems (e.g., three-dimensional imaging with synthetic aperture radar) requires the coordination of multiple sensing platforms. Consider a swarm of unpiloted air vehicles (UAV's) whose task is to locate and image potential targets hiding under dense foliage. The swarm must achieve three objectives that require different behaviors on the part of individual UAV's.

In *searching*, each UAV must effectively cover a large search space and revisit locations regularly, maximizing detection probability based on known characteristics of the target (e.g., visibility angle), while not exhibiting any obvious systematic search patterns that would permit mobile targets to execute simple avoidance strategies. A single sensor can generate enough information to suggest the presence of a target, though it cannot image the target by itself.

When a vehicle detects a target, it announces the location of the target, and vehicles that receive this announcement begin a coordinated *imaging* task. In

this phase, each vehicle must collect data from varying angles along linear trajectories (box) while minimizing both the effort (the number of required vehicles and the distance they must move) and the data collection time (by collecting data in parallel).

In addition, individual vehicles require periodic refueling or other *maintenance*, and the swarm must ensure that individual vehicle requirements are met without compromising the ability of the overall swarm to continue functioning.

Summary of Architecture. Our stigmergic approach to this problem uses digital pheromones. An important contrast with the pheromone mechanisms in our other two example applications is that while those applications envisioned a network of physical nodes maintaining the pheromone field *externally* to the agents, in this case each agent maintains an *internal* pheromone map that tiles the search space into discrete cells. Each cell is a place in a pheromone infrastructure, which means that the agent that controls the vehicle may deposit and sense digital pheromones of different flavors in that cell. In principle, agents could propagate these maps to one another through local interaction, thus achieving a stigmergic analog to the DAI technique of partial global planning (Durfee and Lesser, 1991), but even without generating such a “global distributed view,” the local iconic representation has significant benefits over more conventional robotic techniques such as occupancy maps.

During *search*, when a vehicle passes through the area in the search space that is assigned to a particular cell, it deposits a unit of the visitation pheromone into that cell in its internal map. In addition, the agent broadcasts its location, and the agents of any other vehicle within communications reach then deposit a visitation pheromone into their maps too. Thus, the agents mark cells that some member of the swarm has already visited. Figure 17.11 shows a snapshot of the visitation pheromone map of one agent in the swarm.

Local concentrations of pheromones lose strength over time, which enables the swarm to “forget” visitations to locations that occurred a long time ago. This knowledge management process ensures that the search process keeps revisiting locations in case targets have moved in.

The individual agent decides its vehicle’s trajectory based on its internal map of visitation pheromones. Once it has reached its previous goal, the agent probabilistically selects a new location. The probability of the selection of a particular location is inversely proportional to its distance to the vehicle’s current location and to the strength of the visitation pheromone concentration in the cell that covers this location. Thus the agents tend to prefer nearby locations that have not been visited recently, and collectively explore the whole search space.

An agent that detects a potential target dynamically forms an *imaging* team. Team formation is a collaborative process in which agents bid for a role in

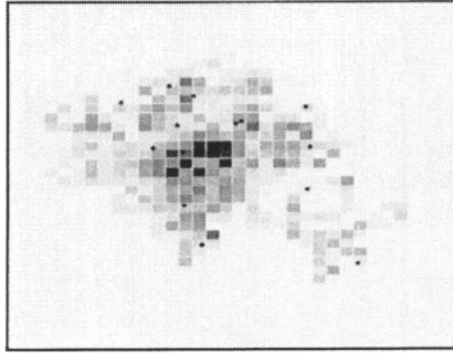


Figure 17.11. Visitation pheromone map of one UAV in the swarm

the team depending on the match of the vehicles' imaging capabilities with the role's requirements (hard constraint) as well as the current distance of the eligible vehicles to the detected target (soft constraint).

Once roles are assigned, the team members plot the optimal trajectories for their respective data acquisition flight and execute the imaging task. Depending on the imaging modality (coherent vs. non-coherent), the data acquisition may be executed individualistically or synchronized across the team. Once the task is completed, the team disbands and the agents resume their search behavior.

A team-based approach to *maintenance* can accommodate UAV's with different fuel consumption rates, as well as variations in the availability of maintenance resources at the base. UAV's deposit a pheromone flavor that communicates the intensity of their current desire for maintenance, while the base propagates a pheromone indicating its current level of load. A UAV's decision to shift into the maintenance role is promoted by its own desire for refuel and inhibited by the level of refueling pheromone it senses from neighboring UAV's and the load pheromone propagated from the base.

Figure 17.12 shows a screen shot of this system. Most of the UAV's are scanning the area in search mode, but four have formed a verification team to image a suspected target, while one is on its way back to the refuel station.

Coupling 1: Use a distributed environment. The environment maintained by each UAV is not distributed, but locality of interaction among UAV's is enforced by their geographical dispersion over the search area.

Coupling 2: Use an active environment. The pheromone environment implements the usual pheromone dynamics of aggregation, propagation, and evaporation. In addition, each UAV's physical environment includes the other UAV's, whose behaviors change based on their individual experiences.

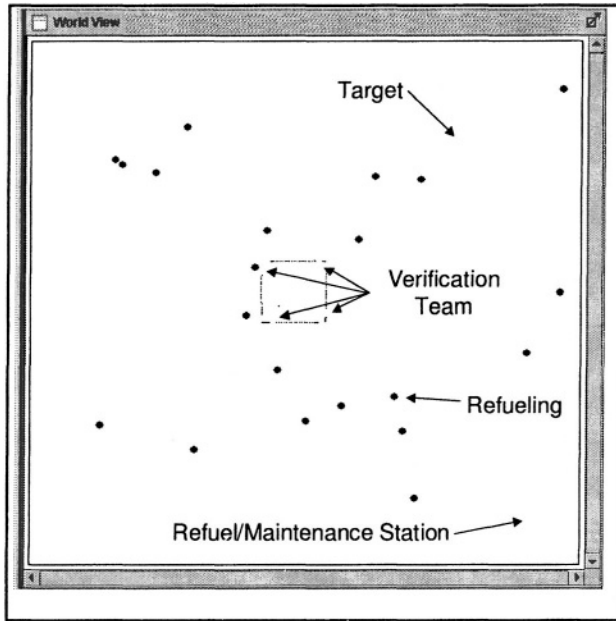


Figure 17.12. Stigmergic role differentiation

Coupling 3: Keep agents small. No single UAV can do the entire task. At least four are needed to image a target, and even more are required to maintain a high level of search.

Coupling 4: Map agents to Entities, not Functions. The agents in this system correspond to physical UAV's.

Autocatalysis 1: Think Flows rather than Transitions. The main information flows in this system are the pheromone flows of Figure 17.8, and the communications flows between a UAV that has detected a target and the other UAV's whom it seeks to recruit to perform imaging.

Autocatalysis 2: Boost and Bound. A UAV's attraction to an imaging team is based on positive feedback, while the visitation map approach to dispersing the UAV's is an example of negative feedback.

Autocatalysis 3: Diversify agents to keep flows going. UAV's are diverse in their location. In addition, each UAV decision (the angle at which to traverse the search area in search mode, whether to join an imaging team, whether to return for refueling) is stochastically weighted.

Function 1: Generate behavioral diversity. Each function requires multiple agents, and each agent supports all three functions. Symmetry among agents is broken by making decisions with weighted stochastic functions.

Function 2: Give agents access to a fitness measure. Several fitness functions influence agent behavior. A UAV's proximity to another that has sensed a target, and its sensory configuration, influence whether it joins an imaging team. Its current fuel level and the load level of the base influence whether it enters maintenance mode.

Function 3: Provide a mechanism for selecting among alternative behaviors. Agents decide whether to image based on a collective sharing of information in a bidding process. Agents decide whether to enter maintenance mode based on their own fuel level and the load at the base.

4.3 Dynamic Target Selection and Path Planning (Parunak et al., 2002b)

The Problem. The current generation of UAV's reduces the threat to human operators, but leaves several problems unresolved.

- It does not decrease the manpower requirements. Each aircraft requires a flight crew of one to three people, so deploying large numbers of UAV's requires committing and coordinating many human warfighters.
- The high-bandwidth needed for linking the flight crew to the aircraft places severe constraints on available communications resources.
- Fusion of information from multiple sources (satellite imagery, sensors on UAV's, unattended ground sensors, information from special forces in the field) is a continuing challenge.

We want a UAV to be able to manage the details of its own mission, avoiding dynamic threats as soon as they arise and planning its path to optimize its movement through the battlespace.

Summary of Architecture. Like our other two examples, this application uses digital pheromones. These pheromones live in a network of *place agents*, which represent regions of the battlespace. All place agents can run on a single computer for simulation purposes, but in actual deployment each place agent might run on an enhanced unattended ground sensor (UGS) placed in the battlespace by air drops or artillery and responsible for any location to which it is closer than any other UGS. We refer to such an enhanced UGS as a HOST (Hostility Observation and Sensing Terminal). Each place agent is a neighbor to a limited set of other place agents, those that are responsible for adjacent regions of space, and it exchanges local information with them. In addition to place agents, the system includes *walker agents*, representing physical resources such as UAV's. Walker agents move through the battlespace by interacting with the place agent for each region that they visit. Place agents and

walker agents are software entities, while HOST's and UAV's are the hardware in which they run.

Each place agent maintains a variable corresponding to each pheromone flavor. It augments this variable when it receives additional pheromones of the same flavor (whether by deposit from a walker agent, from its own sensors, or by propagation from a neighboring place agent). It also evaporates the variable over time, and propagates pheromones of the same flavor to neighboring place agents based on the current strength of the pheromone. Different flavors may indicate the presence of a *threat* that should be avoided in the place's region or the presence of a *target* that should attract UAV's.

The development of a path by a natural ant colony depends on the stochastic interaction of many ants, some of whom wander off and die. Current UAV's such as the Predator and the Global Hawk are far too expensive to use in a stochastic search mode. Instead, each UAV's walker agent periodically emits *ghost agents*, software agents without a corresponding hardware resource. These ghost agents are attracted by target pheromone and repelled by threat pheromones, and lay down a path pheromone to store the results of their explorations. Reinforcement of this path pheromone by multiple ghosts leads to the emergence of a path that the UAV then follows. Recent advances in inexpensive micro-UAV's opens up the potential for having the UAV's themselves swarm, as in the example discussed in section 4.

Figure 17.13 illustrates the functions of the different pheromones in this process. On the left, intelligence about threats is translated into threat pheromones that propagate only a short distance, since their purpose is not to attract distant ghosts, but to prevent nearby ones from wandering into danger. In the center, intelligence about targets results in target pheromones that propagate widely, attracting ghost agents. The higher-priority target (to the west) emits pheromone at a higher rate, thus generating a broader field. The right-hand display shows the path pheromones deposited by the ghost. A UAV following the ridge of this field will be attracted to the appropriate target, while avoiding intervening threats.

Coupling 1: Use a distributed environment. The network of HOST's provides an environment that is physically distributed throughout the entire battlespace.

Coupling 2: Use an active environment. The HOST's implement the pheromone dynamics of aggregation, propagation, and evaporation.

Coupling 3: Keep agents small. Intelligence about the battlespace is not concentrated in a single machine, but maintained across many HOST's, each responsible for a small region. The path planning is done by ghost agents, which are small compared with the UAV's walkers. In our experiments, each walker has about 300 concurrent ghosts.

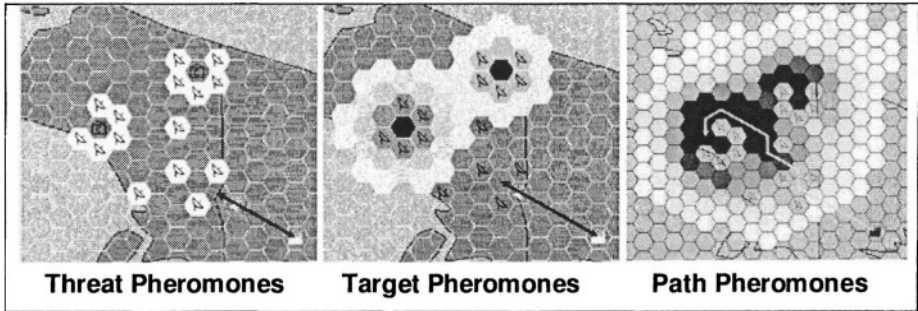


Figure 17.13. Pheromone flavors in emergent path planning

Coupling 4: Map agents to Entities, not Functions. Agents correspond to physical regions and resources.

Autocatalysis 1: Think Flows rather than Transitions. The basic flow is the pheromone cycle of Figure 17.8.

Autocatalysis 2: Boost and Bound. Path emergence among the ghosts is the result of positive feedback as they respond to path pheromones already in place, combined with the bounding influence of pheromone evaporation over time. The ghost population is maintained by continuous birth and programmed death.

Autocatalysis 3: Diversify agents to keep flows going. The system uses three main species of agents: place agents, walker agents, and ghost agents. In addition, different resources have different walkers, different regions have different place agents, and ghosts diversify themselves through stochastic movement. Walkers and ghosts deposit and sense pheromones in the place agents, and thus pass information among themselves.

Function 1: Generate behavioral diversity. The system's main function is path planning, in which all agents participate without any of them dominating. An important class of diversity among ghost agents is the equation by which they translate pheromone levels that they sense in their immediate environment into movement decisions. Originally, we hand-tuned the parameters of this equation. We found improved performance when we allowed the parameters to vary around the hand-tuned mean, and even more improvement when we evolved the parameters (Sauter et al., 2002).

Function 2: Give agents access to a fitness measure. The speed with which a ghost reaches a potential target and returns home is a good measure of the fitness of its search parameters, so we use the lifetime remaining to a ghost as its fitness measure.

Function 3: Provide a mechanism for selecting among alternative behaviors. We use a variety of the genetic algorithm for adjusting the distribution of search parameters in the ghost population. An important characteristic of our application is that this adaptation happens as the system operates, not in an off-line planning process.

5. Conclusion and Prospect

Swarming systems have demonstrated their effectiveness as an alternative model of cognition. This experience is leading to a growing body of engineering knowledge for the deployment of such systems. They are best suited for resource-constrained systems of discrete interacting elements that exhibit distribution, decentralization, and dynamic change. The self-organization that gives these systems their power requires not only interaction among the agents, but the potential for autocatalytic loops, and some mechanism (such as synthetic evolution or particle swarm optimization) for selecting appropriate behaviors from a wider repertoire based on some fitness function. We have deployed these mechanisms successfully in a number of applications, including distributed pattern recognition, team formation and management, dynamic target selection and path formation, resource allocation, document search and retrieval, and ecosystem management.

This engineering perspective on swarming systems recognizes that for some applications or problems, conventional cognitive techniques may be more appropriate. Now that we understand where swarming systems are appropriate and some of the principles that enable them, the next challenge is integrating them with more conventional cognitive systems. We are pursuing several lines of research in support of hybrid agent systems, including

- Using swarming systems as internal “brains” for more conventional cognitive systems;
- Integrating fine-grained and coarse-grained agents as peers in a single system, with fine-grained agents providing ease of implementation and reduced need for knowledge engineering, while coarse-grained agents provide a clearer cognitive interface to human stakeholders;
- Developing mathematical methods for imputing cognitive behavior to non-cognitive agents in support of integration with cognitive agents; and
- Developing a design and specification methodology at a sufficiently abstract level that it can be applied to either class of agent.

Acknowledgments

This work is supported in part by DARPA under the JFACC program and WASP seedling. The views and conclusions in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

Chapter 18

ONLINE ENGINEERING AND OPEN COMPUTATIONAL SYSTEMS

Martin Fredriksson and Rune Gustavsson

Abstract We strongly believe that agent-oriented approaches to system development come with a natural level of abstraction and therefore have something valuable to offer. However, in doing so, any comprehensive agent-oriented methodology necessarily has to be grounded in issues and solutions of relevance in contemporary research and development areas such as Grid computing and autonomic computing – in order to realize the visions of ambient intelligence. Current efforts of AOSE, however, mostly focus on traditional methods of software development, provides implementations of stand-alone agent systems, or isolated experimental platforms. These efforts are, of course, worthwhile in themselves but have clear limitations when it comes to their contribution and fulfillment of visions such as ambient intelligence. Consequently, in this chapter we introduce the methodological approach of online engineering. As such, this methodology has explicitly been designed to meet what we conceive as the major challenges and limitations in contemporary approaches of AOSE. In fact, we argue that these limitations primarily are due to a strong focus on current practice in software engineering, rather than on engineering of grounded open computational systems. In this respect, online engineering provides us with the models, methods, and tools to facilitate the necessary transition from programming of abstract machines towards development of grounded physical systems, e.g., from software engineering to engineering of open computational systems.

1. Introduction

MAS have during the last decade been a very active area of research, development, and experimentations (Parunak et al., 1998b; Parunak et al., 2000). The attractiveness of these efforts is mainly grounded on the natural high-level description power of agents in systems analysis and design. In particular, this is also the case when it comes to describing future complex information systems, comprised by networks of interacting agents (people and software agents). Hitherto there has, however, mainly been a research push in efforts to find methodologies and techniques that bring the inherent possibilities of

agent approaches into an engineering discipline of future complex systems, e.g., ambient intelligence and information ecologies (Gustavsson and Fredriksson, 2003). With only a few exceptions, the research push of MAS as a viable platform for (top-down) understanding, design, development, and maintenance of future ambient intelligence systems has hitherto almost been isolated from industry's (bottom-up) focus on next generation of network enabled capabilities, e.g., peer-to-peer computing (Loo, 2003), Grid computing (Foster et al., 2002) (see chapter 20), and Web services (Stafford, 2003).

Recently there are, however, some efforts to combine the ideas from MAS research with the technological push by industry as mentioned above. Examples include agent technologies combined with (Web and Grid) services and semantic net ideas. However, we claim that most of those efforts are rather straightforward extensions of ideas from object-oriented software engineering, e.g., DAML and AUML (Huget, 2002a) (see chapter 12), and AOSE, e.g., Gaia (see chapter 4). With respect to crucial issues of initiatives such as ambient intelligence, these approaches all come with their inherent limitations. In parallel, there is an industrial push towards high-level system methodologies that address issues of relevance in ambient intelligence, e.g., IBM's effort on autonomic computing.

In essence, we strongly believe that agent-oriented approaches to system development come with a natural level of abstraction and therefore have something valuable to offer. However, in doing so, any comprehensive agent-based methodology necessarily has to be grounded in issues and solutions of relevance in Grid computing and autonomic computing – in order to realize the visions of ambient intelligence. Current efforts of AOSE, however, mostly focus on traditional methods of application development, provides implementations of stand-alone agent systems, or isolated experimental platforms. These efforts are, of course, worthwhile in themselves but have clear limitations when it comes to their contribution and fulfillment of visions such as ambient intelligence.

Consequently, in this chapter we introduce the methodological approach of online engineering. As such, this methodology has explicitly been designed to meet what we conceive as the major challenges and limitations in contemporary approaches of AOSE. In fact, we argue that these limitations primarily are due to a strong focus on current practice in software engineering, rather than on engineering of grounded open computational systems. In this respect, online engineering provides us with the models, methods, and tools to facilitate the necessary transition from programming of abstract machines towards development of grounded physical systems, e.g., from software engineering to engineering of open computational systems. To that end, we have to reassess and specify what we mean by open systems, validation, the crucial difference between formal semantics and behavior semantics, as well as tools enabling

maintenance of systemic properties such as sustainability and dependability. We will address some of these issues in the material presented in this chapter.

In section 2 we explain the main philosophy behind the approach advocated herein. Furthermore, the basic framework and model of open computational systems is also discussed in this section. In section 3 an outline of the actual method applied in online engineering is introduced as well as technologies used in experimentation with online engineering and open computational systems. In section 4 we introduce a number of system prototypes that have been used to validate the overall methodological approach advocated herein. Finally in section 5 we summarize issues to be addressed by means of future research and development activities.

2. Open Computational Systems

The Turing machine is the mathematical model of computing behind methods and techniques for software design. In effect, computing is modeled in terms of programming (implementing an algorithm on) a Turing equivalent abstract machine. Models, methods, tools, and techniques for design, verification and validation, as well as implementation of algorithms to be applied in some particular problem domain are therefore key areas of computer science and software engineering. In the early days of computing the problem domains were mainly numeric calculations of a scientific or administrative nature. In essence, there was (or is) no semantic problems in these areas of applicability since the implementation of numeric calculations can be done in a syntactic way (symbol processing with a formal semantic) and the interpretation of the results is therefore straightforward. Furthermore, the applications were running on stand-alone machines. Therefore, if the algorithms were correct, the implementation valid, and the hardware was working we would be done. In essence, we could do a lot of the hard work (design) offline and focus on implementation and testing (based on the algorithm design) to ensure a quality product. Today we have advanced methods and tools supporting the software lifecycle, e.g., object-oriented approaches related to the unified modeling language. The information systems of tomorrow, e.g., ambient intelligence, are however of a quite different nature. The machinery is networked, the individual components are semantically rich, and the systems' behavior is intrinsically dynamic – they are in every respect open.

In effect, dealing with open systems means that we cannot model all possible events of interaction between system entities (software) in beforehand (offline). Hence, we cannot model or validate the actual behavior and semantics of ongoing processes offline. Therefore, we must better understand the implications and impact of open systems, i.e., we cannot rely on offline validation of individual components to infer a globally coherent online system behavior.

Instead, we need an approach that explicitly addresses validation of systems and their inherent structures and processes undergoing continuous evolution. In the following material, we therefore introduce a general outline of such a methodological framework as well as a related model, which we have denoted *Visions of Open Computational Systems* (VOCS).

2.1 Framework

Above all, the notion of open computational systems addresses the issue of programming physical systems, as opposed to programming abstract machines. To study open computational systems therefore requires us to focus on issues relevant in observation and instrumentation of physical phenomena:

- What is the primary stimulus of system dynamics?
- What is the primary catalyst of system dynamics?
- What is the primary isolator of system dynamics?

Investigations of open systems have been fundamental in our scientific understanding of nature. An important model of openness in natural systems considers the dimension of energy flow as the primary source of evolving system states. Such a source of evolution, or stimulus, is considered to be transferred back and forth over the physical space between two distinct entities. In this respect, chemical compounds and biological organisms are considered as open but a transportation vehicle is not. As such, open computational systems are physically grounded systems of interacting entities, but instead of considering a flow of energy as the primary dimension of system evolution, we consider a flow of information.

Dynamics in open computational systems is a direct result from entities that transfer information between each other over some physical space. However, as opposed to the primitive entities of chemical compounds and biological organisms, the event that triggers an interaction between two entities is not due to the laws of nature. Instead, the catalyst of dynamics in open computational systems is observation of physical phenomena. However, to make observations of physical phenomena necessarily requires the capability of isolation.

The perhaps foremost issue of concern in investigations of dynamical systems would be the identification of a particular system's bounding space – isolation. However, we argue that the boundaries of systems in nature necessarily must be considered from two different perspectives – physical as well as cognitive. That is, the boundaries of some particular system are primarily a cognitive construct that lies in the eyes of some beholder. Furthermore, as opposed to abstract machines, physical systems can be observed by an infinite set of beholders. In this respect, the boundaries of some system are always

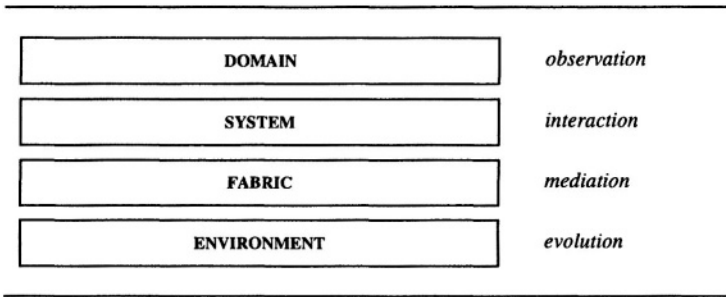


Figure 18.1. The model of open computational systems focuses on the transition from (implicit) models of symbol manipulation towards (explicit) models of system interaction

possible to define as soon as a cognitive entity has the capability to observe the system. Also, the identification of a system's physical boundaries is therefore solely dependent on a particular observer's perception of its cognitive boundaries. Consequently, a physical system can only be isolated for further studies if an observer first articulates the system's cognitive boundaries and then use these to identify its physical boundaries.

2.2 Model

A key aspect of open computational systems is their characteristic property of being open and dynamic in nature. We consider such systems as situated in physical environments where new cognitive entities can appear and disappear at will as well as observe and interact with each other. Consequently, as previously indicated, a model of open computational systems should address three different primitives of system dynamics: information exchange (stimulus), environment observation (catalyst), and cognitive bounding space (isolator). However, it is equally important that such a model also addresses a separation of concerns regarding both theory and practice of open computational systems. That is, our model should be equally applicable in scientific investigation as in system development. Therefore, we propose a model of open computational systems that is structured as follows (see Figure 18.1):

- Environment;
- Fabric;
- System; and
- Domain.

The first layer of our model is that of the environment of evolution. From a modeling perspective, the notion of a system's physical environment is perhaps the most fundamental aspect as it explicitly aims at identifying the primary source of system evolution. That is, if validation of system behavior is our ultimate goal, explicating a physical environment and its ramifications will increase our understanding of potential factors that might shape the behavior of some particular system under study. The influence of nature should not be underestimated – physical systems are governed by laws that necessarily have precedence over logical rules and intentions.

The second layer of our model is that of the fabric of mediation. In the physical environment of an open computational system there exists a dynamic network of interconnected computers. We consider the primary role of each and every node in such a network to be that of mediation, i.e., to facilitate the existence of system entities as well as their observation of and interaction with the physical environment. That is, the fabric layer of our model addresses the aforementioned primitives of stimulus and catalyst.

The third layer of our model is that of the system of interaction. This is where we consider the actual behavior of open computational systems taking place. It is at this layer that system entities make their environment observations which, in effect, can result in information exchange. As such, the processes that take place at the system layer should be considered as the actual behavior of an open computational system. The fourth layer of our model is that of the domain of observation. Since a computational entity cannot sense a physical environment by the same means as a human agent, our model of open computational systems necessarily must provide for such isolation of physical phenomena by some other means. Instead, we can provide for support of isolation at the fabric level, by means of information structures at the domain layer which are related to and accessible by entities at the system layer.

3. Online Engineering

As previously indicated, the notion of open computational systems should above all be considered as a methodological transition from programming of abstract machines towards programming of physical systems. At first, this change in focus introduces us to issues of a somewhat theoretical nature. Previously, we have argued that any model of open computational systems necessarily must consider their physical bounding space (environment) as well as their cognitive bounding space (domain). In effect, these two aspects of the model aim at system isolation. Furthermore, as a fundamental constituent of some physical bounding space, the model considers a network of mediators (fabric) that supports an ongoing process of interaction between entities (system). Now, with such a model at hand, we can address the notion of open compu-

tational systems from a practical perspective and focus on the actual problem at hand – validation of system behavior. In doing so, we emphasize the need for new methods, platforms, and instruments to be used in scientific investigation and development of open computational systems. Standard techniques for (offline) software engineering must be combined with models, methods, and tools to support online articulation, construction, observation, and instrumentation of open system – embedded in physical environments. The method also enables us to validate the proper behavior of open computational systems, i.e., control in processes of evolutionary system development. We have denoted this method *Online engineering*.

3.1 Method

As opposed to natural systems, the constituents of an open computational system do not come into existence as a result from the physical processes of nature. Furthermore, they do not lend themselves to cognitive inspection as systems in nature do. Consequently, with the ultimate goal of validating behavior in such systems, we have devised a method that emphasizes the previously outlined model of open computational systems. Furthermore, it should be noted that the basic rationale of this method is that we consider information systems of the future so complex that validation of system behavior, according to formal specifications, will not suffice. Instead, validation of system behavior must be performed as an iterative process where a continuously evolving system can be trialed online according to its physical specifications. We consider this iterative method of validation in terms of four activities (see Figure 18.2):

- Articulation;
- Construction;
- Observation; and
- Instrumentation.

According to our previously outlined model, we must first articulate the structures present at the first three layers of an open computational system. This activity must necessarily start with the identification of a system's environment and corresponding ramifications. As such, the articulation of the physical environment will then facilitate the identification of the involved fabric entities, i.e., the physical artifacts that implement sensor and actuator capabilities but above all mediation capabilities. Finally, according to the application domain that has guided us in articulating the previous layers, we must identify the entities that supposedly should implement the system behavior we are addressing.

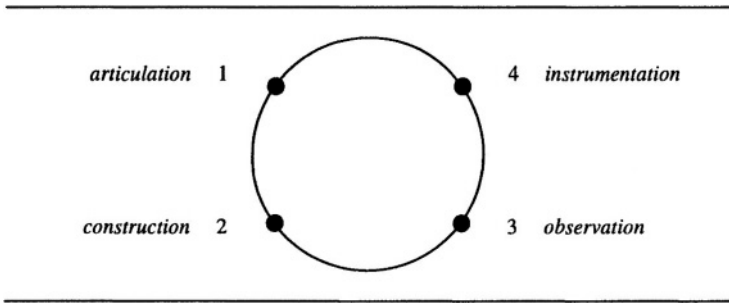


Figure 18.2. The method of online engineering focuses on the transition from offline methods of articulation towards online methods of instrumentation

Once the environment, fabric, and system layers have been articulated it is time to construct (i) the fabric entities; (ii) the system entities; and (iii) the domain entities. However, at this point it should be noted that the initial articulation of domain entities as such is done as part of constructing the fabric and system entities, i.e., the actual construction of domain entities is performed by the fabric and system entities themselves. When the construction phase of our method is finished, all fabric and system entities are launched and their combined behavior will emerge – readily available for observation.

Consequently, the offline activity of engineering will now go into its online phase. By means of the constructs present at the domain layer, not only will we be able to isolate all entities present, so will all of the involved entities. That is, by means of observation, both human agents as well as system entities can isolate physical phenomena in the involved environment. This capability of observation is, as previously indicated, the fundamental catalyst of behavior in open computational systems. However, it is also the fundamental basis of manual as well as automated system validation and, consequently, instrumentation.

At the heart of online engineering is the validation and establishment of a stable system behavior. As such, the method therefore emphasizes an iterative process where real time observation of systemic qualities is of the essence. Furthermore, if such observed systemic qualities indicate that the behavior of some particular system is unstable; we should be able to instrument the system in such a way that the system can be kept online – a complete shutdown should be considered as a complete failure (Fredriksson et al., 2003). In this respect, if a negative system quality has been observed, the four phases of online engineering is simply performed once again. However, it should be noted that in such a second iteration of online engineering both human operators as well as system entities can partake.

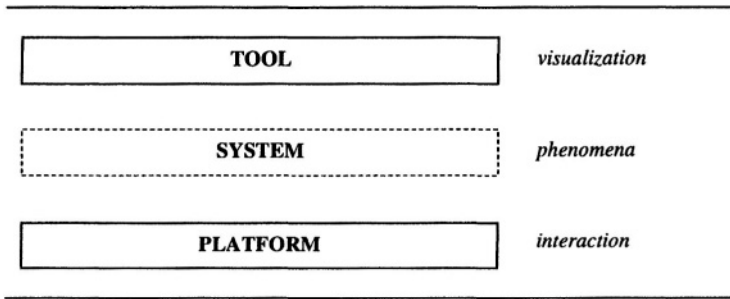


Figure 18.3. The technologies of online engineering are specifically tailored towards interaction with and visualization of complex phenomena which are a priori unknown.

3.2 Technology

The model of open computational systems and the method of online engineering are only two out of several methodological components. In this respect, they need to be complemented in terms of technologies that support practical experimentation and actual experience of some particular phenomenon of interest. Consequently, we introduce the methodological component of technologies (see Figure 18.3):

- Platforms; and
- Tools.

SOLACE. In practice, the foremost requirement of open computational systems is their need for platform support of online articulation and construction. Furthermore, support of such mechanisms enables us to observe constructs and behavior of open computational systems as well as to instrument such behavior. In order to identify and experiment with such mechanisms, we have developed a distributed *Service-Oriented Layered Architecture for Communicating Entities* (SOLACE). Contemporary tools and approaches toward support of behavior in open computational systems need to address the pivotal notions of physical scalability and conceptual decoupling. In effect, our platform should therefore be considered as the supporting infrastructure that manages the primitive dimensions of scalability and decoupling of open computational systems. In principle, this platform explicitly supports the following abstraction layers, and their inherent dynamics, of open computational systems: interaction fabric, system behavior, and cognitive domains.

VIRTUE. An appropriate representation of qualitative aspects in research and development of complex systems is of the essence. This applies to complex systems in reality as well as systems of a virtual nature. In effect, when

we are dealing with complex systems of a virtual nature, such as open computational systems, we need the practical support in experiencing virtual phenomena. Examples of such experience classes would typically correspond to visual and aural effects, as well as interaction with network-centric constructs. Consequently, in order to provide for such support, we have developed a *Virtual and Interactive Real Time Universe Engine* (VIRTUE). In principle, this platform provides for real time interaction and rendering of visual, aural, and network phenomena of a virtual nature.

DISCERN. The requirement of platform support for online engineering is primarily introduced by the open computational systems as such. However, when it comes to observation of behavior in these complex systems, it is important to remember that, as human agents, we need specifically tailored instruments and tools to do so. In practice, the activity of observation aims at acquiring certain quantifications of some particular domain's characteristic qualities. By means of instruments, we can also acquire such quantifications and use them as a basis for system modifications. However, we also need to visualize the acquired information in a tractable manner. Consequently, we have developed a *Distributed Interaction System for Complex Entity Relation Networks* (DISCERN), which explicitly addresses dynamic and real time observation, visualization, and interaction in open computational systems.

4. Methodological Benchmarking

Let us return to the initial problem addressed in this chapter; a methodological approach that would allow for joint ventures between academia and industry in pursuit of a transition from programming of abstract machines towards development of grounded physical systems. We have previously indicated that an applicable approach necessarily must not consider one explicit methodological component in isolation, but rather that the issue must be addressed by means of a comprehensive methodology. In doing so, we have outlined the framework and model of open computational systems as well as the method and technology of online engineering. However, advocating the applicability of comprehensive methodologies must necessarily also survive practical trials – benchmarking. In an attempt to evaluate the applicability of our methodological approach of online engineering and open computational systems, we have therefore developed a number of benchmark prototypes. In essence, these prototypes have been specifically tailored to address certain application domains of interest to society at large but also of academic and, above all, industrial relevance. It should however be noted that an in-depth discussion regarding these benchmarks is out of scope in this particular chapter. Still, in the following material we introduce three prototype systems that, respectively addressed the following application domains (see Figure 18.4):

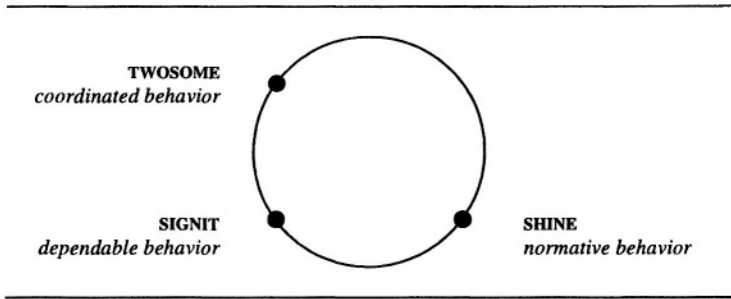


Figure 18.4. The methodology of open computational systems and online engineering has been trialed in terms relevant (scientific and industrial) issues of ambient intelligence

- Network-centric warfare and coordinated behavior;
- Critical infrastructures and dependable behavior; and
- Distributed healthcare and normative behavior.

TWOSOME. Research and development of information systems for defense and warfare have changed most dramatically during the last decade; from weapons of mass destruction to sustainable systems of coordinated and computationally empowered entities, i.e., network-centric warfare (Alberts et al., 2001). From a holistic perspective, the involved systems are comprised by a wide range of services: sensor and actuator systems, detection and weaponry systems, as well as communication and support systems. In effect, the behavior of each and every system component, as well as their synthetic behavior, has to be dependable and trustworthy in operations under dynamic and hostile conditions, and perhaps even more challenging; the notion of system lifespan has to be considered in terms of decades. Consequently, at the core of a prototype system for *Trustworthy and Sustainable Operations in Marine Environments* (TWOSOME) is the development of a MAS, where interacting and coordinated entities and services temporarily come together in a physical setting; in order to perform a particular assignment under dynamic and hostile conditions. In essence, the system developed is subject to a validation of qualities such as information fusion, coordination, and adaptation. In this particular system there exist two interrelated application domains and, consequently, missions involved: creating and removing physical threats at some particular environment location. In effect, the various agents involved are positioned in a physical environment and therefore, by means of their autonomous and cooperative behavior, create an evolving state of affairs that is difficult to handle by means of a single coordination mechanism. In essence, an attacker is positioned in the environment and set to detonate whenever the presence of a particular vessel

is identified in the surroundings. Correspondingly, three coordinated defenders are assigned the role of sweeping different environment locations and, in effect, aim at removing potential attackers. However, due to potential degradations in functionality, the defenders must continuously reassess their current situation and take actions accordingly (Fredriksson and Gustavsson, 2003).

SIGNIT. Important issues related to Europe's future energy systems involve safety critical infrastructures and environmental concerns. In addressing such concerns, it is important that an accessible scenario is identified and constructed at an early stage, i.e., in order to provide for the basic setting of future experimentation and empirical investigations in the domain of critical infrastructures. Therefore, in order to account for such an experimental setting, we have developed a prototype of *Sustainable Infrastructures and Geospatial Networks of Information Technology* (SIGNIT). The prototype addresses the physical environment of Öland, at the south-east coast of Sweden, and two different infrastructures involved: information and energy networks. At each node in these geospatial networks, certain information is accessible by means of localized services, which in turn can be used in online engineering and experimentation with sustainable behavior of critical infrastructures. Services populating the information and energy networks, e.g., production, consumption, and load balancing services, are in focus in this prototype system.

SHINE. Treatment of elderly people and citizens in need of professional care is one of the most important aspects of any society to consider, i.e., proactive support for citizens' quality of life. However, since the involved organizations to a great extent are of a centralized nature, patients need to transport themselves to the hospital in order to receive treatment. If the organizations could be decentralized and still reach out to patients in need of treatment, this would be of great benefit for all of the involved parties. Consequently, we therefore focus on the needs of mobile home support teams in distributed health care organizations, by means of technologies that enable mobile access and peer-to-peer connectivity. Consequently, we developed a system for *Sustaining Health and Interaction in Networked Environments* (SHINE), where interacting and coordinated entities and services temporarily come together in a physical setting; in order to perform sustainable and effective behavior under dynamic and life-dependent conditions. In essence, the system developed is subject to a validation of qualities such as information fusion, coordination, and normative constraints.

5. Concluding Remarks and Future Work

Above all, the notion of online engineering addresses the issue of programming reliable systems, as opposed to programming abstract machines. As such,

the approach of online engineering requires a comprehensive methodology that embraces the concerns of both science and engineering. That is, a new set of models must be developed that emphasizes certain key concepts and their interdependencies – aiming at scientific principles of reliable behavior in open computational systems. In this respect, engineering must develop a new set of platforms and instruments to allow for the construction of reliable systems as well as scientific experimentation with open computational systems.

Coming back to the main issue of online engineering, it is important that the new models of behavior in such systems are identified and trialed as a joint venture between the science and engineering communities. That is, the models of scientific investigation in the area of agent systems must be equally applicable in systems engineering and vice versa. In this chapter, we have therefore presented a first outline of a comprehensive methodology that encompasses both theoretical aspects, i.e., framework and model of open computational systems, as well as practical concerns, i.e., method and technology of online engineering. However, advocating a comprehensive methodology is a quite futile task if the methodology as such is not trialed in terms of actual applicability. Therefore, we have introduced a number of relevant application areas of agent systems, i.e., network-centric warfare, critical infrastructures, and distributed healthcare. These areas of applicability have been instrumental in our development of several system prototypes which, in effect, have been used to identify issues and opportunities of our methodological approach.

With respect to the areas of applicability for agent systems, in combination with the system prototypes developed, we have identified certain concrete issues that have to be addressed by means of future research and development activities. In principle, these activities are all related to the overall goal of realizing the visions of ambient intelligence, but should all be considered from the perspective of a methodological approach of equal relevance to the scientific as well as industrial communities.

The first issue identified in our methodological benchmarking is related to the notion of a framework. As such, the framework should provide the (scientific and engineering) practitioner with a clear and unambiguous world view, i.e., a grounded philosophy that actually encompasses some particular phenomena under study. In our case, the major insight has been the need for a framework that explicitly addresses the transition from programming of abstract machines towards sustainability of physical systems.

The second issue identified involves the notion of models. At the very core of any model is the overall goal of isolating a particular phenomenon, or class of phenomena, and to represent its essential nature in a clear and abstract manner. In the case of models for open computational systems, our major insight is strictly related to the need for a model that addresses the transition from

(implicit) models of symbol manipulation towards (explicit) models of system interaction.

The third issue identified in our methodological benchmarking is related to the notion of a method. As previously indicated, ambient intelligence and open computational systems presupposes the emergence of such complex behaviors that control cannot be fully designed and programmed in an offline manner, but rather that this has to be achieved in an evolutionary fashion. In principle, this means that methods of science and engineering have to consider a transition from offline methods of articulation towards online methods of instrumentation.

The fourth issue identified involves the notion of technology. As is the case with all of natural science, computer science and software engineering is utterly dependant on support from specifically tailored technologies. However, technologies must necessarily be considered as the means to fulfill some particular needs. In our case, dealing with the complexity of ambient intelligence and open computational systems, we need to explicate the need for technologies that are specifically tailored towards interaction with and visualization of complex phenomena which are *a priori* unknown.

Acknowledgements

The authors would like to acknowledge the substantial development efforts provided by personnel at *Societies Of Computation LABORatory* (SOCLAB). Furthermore, it should be acknowledged that research and development of the various technologies and systems outlined herein was funded by the following research projects: *Service-Oriented Trustworthy Distributed Systems* (SOTDS), *distributed intelligence in CRitical Infrastructures for Sustainable Power* (CRISP), and *SUstainable Mobile Services with healthcare applications* (SUMS).

VI

EMERGING TRENDS AND PERSPECTIVES

Introduction

The potentials of agent-based computing in several application areas have been already outlined in the introductory chapters of this book, as well as here and there in various chapters of it. The key message is that agent-based computing, as a novel software engineering paradigm, promotes the design and development of more robust and adaptive applications than traditional software engineering approaches.

The generality of the above consideration is testified by the fact that agent-based software development has already been successfully applied in areas such as Web-based information systems, workflow systems, and manufacturing systems. That is, all those software systems that were traditionally designed and developed using traditional techniques – i.e., object-based and component-based – can provably take advantage of the agent-based approach and of the techniques and methodologies of agent-oriented software engineering.

However, the ICT scenario is continuously changing, and brand novel application areas are emerging, raising very complex issues to be faced by software engineers and developers. For these novel areas, the problem of verifying whether agent-based computing will be a better approach than traditional approach one will not apply. Simply, the general understanding is that agent-based computing will be the only suitable approach to face the intrinsic complexities of these novel applications areas.

The emerging application areas we are referring to are global computing and pervasive computing. The former, global computing, include applications that will execute at a world-wide scale, and will be made up of a large amount of interacting processes/agents, distributed in the Internet and accessing data and resources from wherever in it. The latter, pervasive computing, considers the presence of wearable computers and of an increasing number of computing devices embedded in our physical environments and artifacts. This pervasiveness of computing devices (capable of interacting with each other in mobile wire-

less networks) will provide for enriching our capabilities of interacting with the environment and, viceversa, will make it possible for our environments to become interactive. In other words: our capabilities to compute and access distributed computational resources will become ubiquitous; and the objects and materials that compose our home and working environments will become somewhat smart, intelligent.

The first two chapters of this part focus on two specific aspects of the global computing and of the pervasive computing scenarios, namely the Grid and ubiquitous computing (the scenarios of computational economies and smart materials have already been somewhat discussed in chapters 10 and 15, respectively), and will discuss the role that agent-based computing and agent-based software engineering techniques will play in them. In particular:

- Chapter 19, “Agents for Ubiquitous Computing” by Zakaria Maamar, Walter Binder, and Boualem Benatallah, discusses what can be the value added by software agents to ubiquitous computing, e.g., as a way to help coordinate the large amount of mobile computing devices that will populate ubiquitous computing environments. In addition, the chapter discusses what specific agent-oriented software engineering approaches are needed in the development of ubiquitous computing applications.
- Chapter 20, “Agents and the Grid” by Luc Moreau, Michael Luck, Simon Miles, Juri Papay, Keith Decker, Terry Payne, argues that Grid applications very strongly suggest the adoption of agent-based computing, and review the key uses of agents in Grid applications. Following, the chapters focuses on the specific issue of service discovery in Grids, that the authors identify as a prime candidate for the use of agent technology in Grid applications.

In addition, this part includes the chapter “Roadmap of Agent-Oriented Software Engineering” by Zahia Guessoum, Massimo Cossentino, Juan Pavon. The goal of this final chapter is to summarize the key achievements so far in the area of agent-oriented software engineering, and to use this summary to identify further promising research directions for the near and the long term, so as to sketch a sort of roadmap for the researches in the area. The sub-title of the chapter emphasizes the fact that the chapter itself is the result of a collective work from European researches having participated to the activities of the AgentLink SIG on Methodologies and Software Engineering for Agent Systems. Consequently, what is expressed in the book can, to some extent, reflect the broader perspective acquired during the SIG discussions.

Chapter 19

AGENTS FOR UBIQUITOUS COMPUTING

Zakaria Maamar, Walter Binder and Boualem Benatallah

Abstract Nowadays, advances in location sensors, wireless communications, and global networking are advancing and deploying the ubiquitous computing vision. Ubiquitous computing aims at making computer use invisible to users. Besides the central role that hardware infrastructure plays in the expansion and penetration of ubiquitous computing, other issues still need to be tackled. In this chapter, we discuss the issue of the value-added of software agents to ubiquitous computing. Indeed, software agents have a role to play such as, for example in coordinating the large number of computing devices, whether fixed or mobile, that will populate ubiquitous computing environments. Most importantly, agents in ubiquitous computing need to be context aware so they can adjust their behavior to different situations. Therefore, new agents engineering approaches are needed.

1. Introduction

The development of new computing and communication devices and the increased connectivity between these devices thanks to wired and wireless networks are enabling various opportunities for people to perform their operations anywhere and anytime. Samples of devices are multiple and include just citing a few desktops, cell-phones, and personal digital assistants. Because of the current trend in the acceptance rate of the devices whether fixed or mobile by the user community, it is expected that these devices will become so pervasive that most users will take them for granted. Generally known as Ubiquitous Computing (UC), this vision is the object of several research efforts going back to the statement of M. Weiser about UC (Weiser, 1991). The vision of UC is to push computational services out of conventional desktop interfaces into environments characterized by transparent forms of interactivity (Abowd, 1999). Major hardware developments as well as advances in location sensors, wireless communications, and global networking are advancing and implementing Weiser's vision towards an UC world (Saha and Mukherjee, 2003).

Despite the growing interest in UC, there is still some progress to achieve before UC shifts from the investigation mode to the commercial mode. The support technology, however, is improving at an impressive pace. Most of the research and development efforts are aiming at improving the devices themselves and the technologies these devices will use to communicate. With regard to research on devices, the focus is being on offering further functionalities to users, while reducing size, cost, and power requirements. Since 2001 Java-enabled phones are available on the market; it is now possible to download Java applets from servers to be run on such phones. Accessible, personal, and location-aware are among the main advantages that wireless devices over fixed devices provision (Yunos et al., 2003). With regard to research on communication, the focus is being on improving bandwidth and throughput and on developing new coverage and recovery techniques. At present, the main use of mobile devices is still voice-oriented, but several indicators show that this is changing. Third generation networks (e.g., GPRS: General Packet Radio Service¹, and UMTS: Universal Mobile Telecommunication System²) and recent development of communication and presentation protocols (e.g., XML: eXtensible Markup Language, WAP: Wireless Application Protocol) are being combined to give users a high-quality experience of data-centric services (Ekudden et al., 2001; Ralph and Shephard, 2001; Wieland, 2003).

Besides the central role that hardware infrastructure plays in the expansion and penetration of UC, other issues still need to be tackled to better assist developers of UC applications and potential users as well. Developers are put on the front line of satisfying the promise of businesses and service providers of delivering Internet content to mobile devices. Indeed, since an application for mobile users has different requirements this definitely calls for new techniques to identify and specify these requirements. In addition, because such an application relies on wireless communication channels, the features of these channels have to be emphasized during the design phase. Therefore, new software engineering techniques enhanced with high-level abstraction concepts are required to support developers. With regard to users, it is expected that they will be frequently engaged in complex operations such as searching the net for better business opportunities. Therefore, their association with intelligent components, to act as proxies, is deemed appropriate. Software agents denote these components (Jennings et al., 1998) and can be gathered into groups of agents

¹GPRS is a new non-voice value added service that allows information to be sent and received across a mobile telephone network. It supplements today's Circuit Switched Data and Short Message Service. GPRS has several unique features such as speed and immediacy.

²UMTS is a 3G mobile system being developed by ETSITM within the International Telecommunications Union's IMT-2000 framework. UMTS is an European system which is attempting to combine cellular, cordless, low-end wireless, local area network, private mobile radio, and paging system. It will provide data speeds of up to 2Mbps, making portable videophones a reality.

denoted by MAS. UC environments of the near future will be populated by a large number of computing devices, spread across the network, and often invisible. These devices need to be coordinated for better interactions. Devices, whether carried on by people or embedded into objects, e.g., homes (Edwards and Grinter, 2001), restaurants (Stanford, 2003), will constitute a global networking infrastructure with a new level of openness and dynamics. These interactions raise many new issues that draw on and challenge the field of agents.

One of the main objectives of UC is that devices should provide smart support to users without forcing them to adjust their behavior. Therefore, the access to devices should not be complex despite the large number of functionalities that could be made available. Unfortunately, the opposite often occurs as devices are too complex and people just use the very basic functionalities (Schmidt and Beigl, 1998). An assistant that hides the complexity of all these operations is necessary. Schmidt and Beigl call this assistant an accompanying personal device that eases the interactions with and use of devices (Schmidt and Beigl, 1998). In this chapter, agents correspond to such assistants.

The rest of this chapter is organized as follows. Section 2 provides few examples on the widespread of UC in people's daily-life. Section 3 provides details on the basic concepts that are considered in this chapter including software agents, mobile computing, pervasive computing, and finally ubiquitous computing. Section 4 presents the core components of an UC environment. Section 5 presents the value-added of agents to UC and explains the ways agents can contribute to the development and deployment of UC applications. Finally, section 6 concludes the chapter.

2. Examples on Ubiquitous Computing

UC is attracting the attention of several people from academia and industry as the various projects illustrate (Saha and Mukherjee, 2003):

- **Aura** (Carnegie Mellon University): the objective is to provide each user with an invisible halo of computing and information services that persists regardless of location.
- **Endeavour** (The University of California at Berkeley): the objective is to enhance human understanding through the use of information technology, by making it dramatically more convenient for people to interact with information, devices, and other people.
- **Oxygen** (MIT): the objective is to bring abundant computation and communication, as pervasive and free as air, naturally into people's lives.
- **Portolano** (The University of Washington): the objective is to create a testbed for investigation into the emerging field of invisible computing

that describes the coming age of ubiquitous task-specific computing devices.

- The Active Badge Location System (Olivetti Research).

One of the expected outcomes of UC is to ease the life of people. Various examples have been given such as (i) a refrigerator that knows that milk is running out and thus, needs to be reordered by contacting the supplier; or (ii) a frozen food that transmits the correct duration and power settings to the microwave. These two basic examples identify the key elements that should embody any UC application namely awareness, access, and responsiveness (Fano and Gershman, 2002). For instance, the refrigerator has to be aware of the quantity of milk that is left before it takes any action. Based on that quantity, the refrigerator may access a yellow-pages system to identify which grocery to contact using certain criteria such as proximity or free-delivery. The refrigerator may also have to be responsive to the outcomes of its interactions with the grocery.

Personal Device Assistants (PDAs) and cell-phones are the more “visible” types of devices that constitute most of current UC environments. However, there are many other types of devices that surround people without these ones being aware of their presence (i.e., such devices are unobserved). Today most household appliances have embedded microprocessors. Each one of these small devices offers a specific service to users. Interesting and challenging are the situations when devices will be able to collaborate with each other to build up more complex services.

3. Background

Software Agents. An agent is a piece of software that acts autonomously to perform tasks on users’ behalf (Jennings et al., 1998). The design of many agents is based on the approach that the user only needs to specify high-level goals instead of issuing explicit instructions, leaving the how and when decisions to the agent. An agent exhibits a number of features that make it different from other traditional components including autonomy, goal-orientation, collaboration, flexibility, self-starting, temporal continuity, character, communication, adaptation, and mobility. It is mentioned that not all these characteristics have to be present in an agent. For illustration purposes, certain features are explained below.

- Autonomous: an agent is able to take initiatives and exercise a non-trivial degree of control over its actions.
- Collaborative: an agent does not blindly obey commands, but can modify requests, ask for clarifications, or even refuse to satisfy certain requests.

- **Flexible:** an agent's actions are not scripted; it is able to dynamically choose which actions to invoke and in what order, in response to the status of the external environment.
- **Mobile:** an agent transports itself from one host to another across multiple computing platforms where it resumes its operation.

(Chen and Finin, 2002) note that many of the existing research projects on agents assume that agents will conduct their activities in a controllable environment. In this environment, agents can have a complete knowledge and unlimited resources. However, the authors questioned whether these assumptions are still valid in UC environments that are open, dynamic, and constituted of several interdependent but autonomous components. This definitely calls for new agent-based engineering approaches for UC environments.

Mobile Computing. Mobile computing refers to systems in which computational components (either hardware or software) change their locations in a physical environment (Huang et al., 1999). (Bellavista et al., 2001) decompose mobile computing into user, terminal, and access mobility. User mobility requires providing users with a uniform view of their preferred working environment regardless of their current location. Terminal mobility allows devices to transparently move and connect to different points of attachment. Finally, access mobility entails the dynamic adaptation of the resources and services to meet the requirements of mobile users and terminals.

Mobile computing is also seen as a way of increasing the capability of users to physically carry on computing devices with them while moving (Lyytinen and Yoo, 2002). As a result, devices are made available to users anywhere and anytime. A major limitation of mobile computing is the no-adjustment of the computing model while users on the move. This is mainly due to the unawareness of the context that underlies a user business-environment (Brézillon, 2003). To handle this limitation, users have to control and configure the applications by themselves – an operation most users do not want to be responsible for.

Pervasive Computing. It is another perspective that makes computers invisible. The core idea is to allow a dynamic deployment of a computing model after assessing the environment (Lyytinen and Yoo, 2002). Pervasive computing goes beyond the realm of personal devices; it is almost any device (cloths, tools, appliances, cars, etc.) embedded with chips can connect an infinite network of collaborative devices. Four aspects motivate the widespread of pervasive computing (Ark and Selker, 1999): computing is extended throughout the environment, users are mobile, information appliances are becoming increasingly available, and communication is made easier between persons, between

things, and between both persons and things. (Saha and Mukherjee, 2003) claim that pervasive computing is a superset of mobile computing. In addition to mobility, pervasive systems call for interoperability, scalability, adaptability, and invisibility support to ensure that users have a seamless access to computing whenever they need it.

Ubiquitous Computing. Weiser defines UC as *the method of enhancing computer use by making many computers available throughout the physical environment, but making them effectively invisible to the user* (Weiser, 1993). UC is considered as an integration effort of mobile computing and pervasive computing (Lyytinen and Yoo, 2002). Indeed, any computing device, while moving with persons, can incrementally build dynamic models of their environment and consequently, configure and adjust in response its services. Research on UC includes various aspects such as technology (small devices, wireless communication, location sensing, etc.), psychology (privacy concerns, attention focus, multi-person interaction, etc.), and design (direct interaction, work patterns, etc.).

Borriello discusses a list of challenges that need to be tackled before UC applications become widely accepted (Borriello, 2002). These challenges are related to the characteristics of mobile devices (e.g., small size, low storage capacity, poor computing performance) and wireless networks (e.g., low bandwidth, low throughput, poor reliability). They are summarized as follows:

- Heterogeneous networks: multiple types of networks (i.e., wired or wireless) and communication protocols to manage networks currently exist. That has been foreseeable since networks and protocols are applied to numerous cases with different requirements and expectations. Therefore, devices will not all have to communicate using the same technique. To handle such heterogeneity, transport protocols that transfer packets from one band to another are needed. In addition, protocols that establish and maintain communications particularly for wireless networks are important. Besides that, the lack of processing performance of most mobile devices influences the selection of the communication protocols. It is known that devices will try to shut down as often as possible for various reasons such as battery use reduction.
- Geographic vs. network topology: it is obvious that a connecting path between devices differs when seen from a virtual or physical perspective. A focus on the geographic proximity rather than on the network topology in UC is preferred. Nearby devices should communicate as directly as possible while maintaining their respective security. Moreover, devices have to know with whom they communicate and how the data exchanged is made secure.

- Short-lived connectivity: because mobile devices depend on batteries for their operation, they will always try to limit their power consumption as much as possible. However, a major aspect of any UC environment is communication, which means a power-consumption increase. Therefore, tradeoffs between power consumption and connectivity will be at the core of many works on UC.
- Evolution of long-lived systems: as more devices will be everywhere deployed, it is unlikely that these devices will be all simultaneously upgraded. Support systems that keep the operation continuity are important. Moreover, these systems have to deal with the heterogeneity and dynamic reallocation of bandwidth. The future will bring many more types of devices dynamically relocated and communicating at shorter range.

4. Dimensions of Ubiquitous Computing

Ubiquitous computing is a new way of deploying devices and offering services independently of location and time constraints. Indeed, any piece of information on the Internet will be made available to users anywhere and anytime. However, because of the heterogeneity and distribution that characterize the current context, it is expected that users will face several obstacles. To this end, support mechanisms are needed at the following levels (Schmidt and Beigl, 1998):

- 1 Since users will be given the opportunity to instantly exchange information and services with other unknown users, identification and interaction mechanisms are required.
- 2 Since users will be localized anytime and anywhere, privacy and security aspects have to be dealt with carefully. Both aspects affect the acceptance rate of UC by the community.
- 3 Since users will be given the possibility to use different types of devices, adjusting the devices to the type of users (e.g., novice, expert) is important³.

³The 3W Composite Capabilities/Preferences Profiles (CC/PP) working group aims at developing standards that will enable providers of services to get the capabilities of devices through their profile (Ralph and Shephard, 2001). Examples of such capabilities are memory capacity, storage capacity, and processing speed. CC/PP is an XML based scripting language that is used for describing the capabilities of a device using RDF. The CC/PP profiles of devices may be available on the devices or on a location in the network, whose URL is known to the device. For some users, revealing even the basic device information contained in a typical CC/PP profile may be a cause for concern.

User Assistance Dimension. A large variety of devices are made available. It is unlikely that users will be familiar with them all. Samples of devices include (Saha and Mukherjee, 2003): traditional input (e.g., mice, keyboards) and output devices (e.g., speakers, light-emitting diodes); wireless devices (e.g., pagers, PDAs, palmtops); and smart sensors (e.g., intelligent appliances, embedded sensors). This diversity is a tremendous burden on users who have to be trained in each device. This should not be the case. Devices should become more responsive and friendly by personalizing their functionalities to users (Nakajima et al., 2003). Therefore, devices should be given the capacity to sense the environment, which will make their functionalities adaptable to the current situation. Adaptability is a key approach to deal with the variety of computing environments and changing settings (Popovici et al., 2003). Indeed, new modes of interaction have to emerge so user movement and proximity of devices for instance can be considered (Esler et al., 1999). The main challenge will be to maintain consistency across devices while adjusting and managing the multiple interfaces of these devices to the current context of use.

To illustrate the importance of assisting users and performing their operations in a transparent way, (Maamar et al., 2004) have conducted some research on hybrid environments that consist of a mixture of fixed and mobile devices. The research has focused on the identification of the devices on which services triggered by users will be performed. The identification was based on the computing capabilities of devices vs. the computing requirements of services. Different types of agents have been devised including user-agents to provide assistance to users and resource-agents to identify devices (i.e., fixed or mobile) seen as computing platforms.

Privacy and Security Dimension. Privacy and security are, obviously, intimately related. Privacy is an integral part of users, whereas security is an integral part of the resources and services users will be authorized to exploit. Unlike security, which deals with keeping information and resources away from unauthorized uses, privacy⁴ gives people the right to control their personal information, including when, how, and to what extent information about them could be communicated to others (Kortuem and Segall, 2003).

Privacy is a major concern for users of UC applications (Abowd, 1999; Satyanarayanan, 2001). Users have to be aware that their steps and actions can be monitored and stored for possible use in the future (may be against them). To overcome this concern, potential users have to trust the system in

⁴In the objective of supporting an automatic assessment of the privacy, P3P standing for Platform for Privacy Preferences is an initiative under the auspices of the World Wide Web Consortium (W3C). The aim of the initiative is to establish machine-readable standards that can be automatically compared to the individual privacy preferences of a user. Using a trusted personal device, sensitive information can be exchanged with the environment only if the recipient matches the privacy preferences of the sender.

use (Schmidt and Beigl, 1998). It is necessary to provide privacy mechanisms that are enough familiar to users so they can easily understand and accept. A user should have the authority to decide which information to share and which to keep secret.

Security in UC applications is differently handled from standard applications (not-ubiquitous) that are usually deployed on wired communication channels and powerful computing resources (Kagal et al., 2001). For instance, logging into computers and domains are no longer valid because UC environments are open and dynamic. This increases security risks and problems with access control as services and resources are being made available to anyone who has a mobile device. In addition, the limitations of mobile devices in terms of processing, memory, and bandwidth capacity are still hindering the deployment of security mechanisms on these devices.

Context Awareness Dimension. The integration of the environment and of the context surrounding users into applications is a good indication to the acceptability and usability of these applications by users (Brézillon, 2003). Devices should sense what the user does need next, what he will probably perform, and what actions can be taken to ease his tasks and anticipate his needs. The computing system should be aware of the user's context not only to respond in an appropriate manner with respect to the user's cognitive and social state but also to anticipate his needs (Siewiorek, 2002).

(Schmidt and Beigl, 1998) claim that a context is more than the current location of the user. The context should integrate the people around the user, the situation (e.g., in a meeting, making a phone call), the environment (e.g., location, temperature, time), and the feeling (e.g., pulse, body temperature). Extra information about the context can make a computing device act and react more promptly and efficiently. Three examples of first-generation context-aware applications are discussed in (Siewiorek, 2002): the notification application that alerts users if they are passing within a certain spatial distance to a task on their to-do list (e.g., getting a fax from the business center); the meeting reminder application that alerts users if they are moving away from the site of a meeting whose location has been changed; and finally the activity recommendation application that recommends possible activities or meetings user might like to attend based on their interests.

To enhance systems with context-aware capabilities, many issues have to be addressed. (Satyanarayanan, 2001) has listed several of them, just to cite a few how is context internally represented? How is this information combined with system and application state? Where is context stored? Does context reside locally, in the network, or both? How frequently does context information have to be consulted? And what is the overhead of taking context into account?

(Kouadri Mostéfaoui, 2003) points out that location is the parameter mostly-used in context-aware applications. She argues that this is not enough for example in service discovery and composition process. Indeed, new types of context parameters are deemed appropriate including user context (e.g., role, identity, location, preferences, social situation, permissions profile), computing context (e.g., network connectivity, nearby resources), time context (e.g., day-time, week, month, season), physical context (e.g., weather forecast, humidity, temperature) and last but not least context history that records details on other contexts for future use. Applications can make use of not just the current context, but also past contexts to adapt their behavior for better interactions with users.

5. Contributions of Agents to Ubiquitous Computing

Agent-based Approaches. (Dey, 2001) argue that one of the major problems in UC applications is the lack of standards that permit their evaluation. We believe that the same argument applies to the design, development, deployment, and maintenance of UC applications. To deal with this lack and provide an appropriate support to developers, there is a trend of considering the available approaches and techniques of standard applications (e.g., service discovery protocols such as SLP – Service Location Protocol, and UDDI – Universal Description, Discovery, and Integration) and adapt them to the UC case. Unfortunately, this adaptation does not seem to be straightforward because of the challenges of UC (Dey, 2001): greater physical space over which the system operates; greater availability of the system over time; and larger number of people the system is supporting interactions with and between. Therefore, instead of adapting approaches and techniques to overcome these challenges, it is important to explore and provide from an agent perspective (because of the focus of the chapter) the support to design and develop UC applications.

To illustrate this lack of standards in UC, this example is provisioned. Predictability is one of the evaluation metrics that can be used (Dey, 2001). This metric presents difficulties when applied to UC applications. Indeed, an application is predictable if the user can determine what the effect/impact of a future input/action would be. This is not simple in UC applications because of their high degree of unpredictability. UC applications have to adapt their behavior according to the context of use. However, a user cannot be aware of all the effects of the actions he performs. These effects keep changing from one context to another.

With regard to the fields of agents and MAS, it is largely accepted that most of the achievements whether at the levels of communication languages, coordination protocols, negotiation strategies, or design methodologies (e.g., Gaia, Tropos, ADELFE, all discussed in this book) have been specifically devised for

standard applications (not-ubiquitous). There is an urgent need to put forward new software engineering approaches and design techniques for agent-based UC applications.

(Esler et al., 1999) mention that the technology and protocols used for implementing agents are becoming better understood. The next step should be oriented towards mobile applications with widely distributed data sources and intermittent connectivity. Initially, the use of agents in building UC applications has to satisfy certain requirements that are (Nakajima et al., 2003):

- Extreme portability: since the devices that contribute to an UC application are extremely heterogeneous, the code to develop agents in charge of these devices should be portable particularly when it comes to making agents roam the network of devices. It is unlikely that a single programming language or operating system will be available for all devices in the near future.
- Uniform behavior: since the diversity of the places in which a user can be (e.g., mall, airport, classroom), agents should enable a uniform way of accessibility to the services regularly-used through an intelligent adjustment. For instance, different user interfaces, each appropriate to a specific context, should be able to handle the same set of services.
- High-level abstract concepts: since the development of UC applications is complex, concepts and approaches that ease this development are needed. Agents as a design paradigm have already shown their benefits in various application domains (Moulin and Chaib-draa, 1996).
- Survival systems: since UC applications will be important in people's daily life, operation continuity and survivability have to be maintained despite security attacks or system crashes. Agents that monitor the normal progress of operations will have to make decisions in case corrective actions are required.

Several concepts and approaches have been developed for standard applications in general and agent-based applications in particular. However, we stated that their straightforward use is not granted in the context of UC applications. Chen and Finin back this statement; they listed the obstacles that need to be considered in case of, for example, teamwork principles would have to be applied to UC (Chen and Finin, 2002). These obstacles are decomposed into three areas. First, with regard to the perception area, the ability to perceive the environment in which an agent resides is of importance so the agent can recognize opportunities and avoid conflicts. However, in an UC environment agents have a limited perception due to physical obstacles or computing limitations that could impede their perception of work. Second with regard to the planning

area, to perform some planning agents need to be capable of constantly acquiring comprehensive knowledge about the state of the world, which includes the state of the team goal, the commitments of individual team members, and the plans that the individual agents have to follow. However, because of the lack of resources on certain computing devices, the difficulty of exchanging plans, and even the lack of a common plan representation, planning can be hindered. Finally, with regard to the mobility of devices area, because computing devices are constantly moving in and out of a network environment, it is important to know their exact location and duration of presence in the environment to ensure the existence of long-lasting collaborations. Ubiquitous agents will enter and leave a community without initial notifications. Tracking them will definitely be important.

Three essential features should embody agents when deployed in an UC environment (Sashima et al., 2002): physically groundedness, context sharing, and device and application independence.

- Physically-groundedness feature should enable an agent that is part of a virtual world to be aware of the context of user who is part of a real world. The objective is how to effectively support users who are immersed in an UC environment. It is known that conventional human interface devices are no longer valid and thus, need to be leveraged. To achieve the physically-groundedness, a spatial information database could be part of the agent architecture. The agents can notice the subtle changes of the environment state by monitoring and detecting the revision of data in the database. Despite the simplicity of this solution, several aspects need to be clarified including how often does the agent browse the database? How much information is needed to be stored? And, which information is considered relevant for storage purposes?
- Context sharing feature should consider the fact that anyone can receive a service anywhere and anytime. This calls for a seamless service provisioning across different areas. To realize this context sharing, a context-aware service could be implemented as a collection of autonomous agents. The agents of a service can communicate with the agents of another service to share information between them. Before such a context sharing can happen in a transparent way for users and in a smooth way for agents, a standardization of the agent communication and knowledge representation mechanisms is important.
- Device and application independence feature should promote reusability of agent-based applications for UC to be developed. Several solutions could be considered such as the use of agents that hide the diversity of physical devices and legacy applications.

Capabilities of Agents. A software agent presents a list of attributes that make it suitable for UC applications. Indeed, there is a natural synergy between agents, entities that are capable of complex, dynamic interactions, and UC applications in which such interactions emerge. UC applications are inherently dynamic with devices continually appearing and disappearing. For instance, devices to take part to an UC application should be (i) constant, always on and running; (ii) presence-aware, alerts to the presence of nearby devices and people; (iii) communicative, able to interact with other collocated devices; and (iv) proactive, able to perform tasks autonomously without requiring explicit user intervention (Kortuem and Segall, 2003). The aforementioned features match quite well the characteristics of what an agent is (see section 3). The autonomy and goal-orientation of an agent match a device's proactivity feature. The temporal continuity and self-starting of an agent match a device's constant feature. The flexibility of an agent matches a device's presence-aware feature. Last but not least, the capacity to communicate of an agent matches a device's communication feature.

Besides these attributes, extra attributes such as mobility, collaboration, and adaptation of an agent have a major value-added to UC applications. A mobile agent is an executable program that can move from a source machine to a target machine where it resumes the suspended operations. Such mobility improves speed, flexibility, and disconnection handling (Lange and Oshima, 1999). Because of their asynchronous execution, mobile agents can avoid long periods of connectivity, reducing network load and thus, saving energy⁵.

An example of the use of mobile agents in UC is illustrated in (Binder and Lichtl, 2002) where an architecture of an autonomous station has been developed. The station does not rely on an external power supply system, but it comprises a unit for the generation of power in order to ensure its autonomy. Further, the station is equipped with application-dependent sensors and actuators, and may be deployed in inaccessible environments. Application services are not hard-coded in the station, instead they are dynamically uploaded on demand. They are charged for using the resources provided by the station. In order to support application upload, transmission of results, and remote system maintenance, the autonomous station is connected to a public or private wireless network. Equipped with the necessary sensors and actuators, the autonomous station can be used for several purposes such as on-demand bus stops and traffic monitoring and tracing.

⁵Besides the multiple advantages that the mobile agent technology provisions, several obstacles hinder the expansion of this technology. For instance, obstacles are not only security-related, but also concerning performance and scalability. A shipped code consumes bandwidth, and there may be a high startup overhead to get a mobile agent up and running due to verification and eventually compilation steps.

With regard to agent collaboration, the satisfaction of a user's request may involve several information resources and devices that both are spread across the network. To make sure that this involvement efficiently happens, there is a need for collaboration strategies. Such strategies could help for instance avoiding conflicts on devices and controlling concurrent accesses to resources. Acting as proxies to users, information resources, and mobile devices, agents implement the collaboration strategies and thus, hide the complexity of this implementation.

We see the use of agents in UC at two different levels. The first level consists of associating users with agents to act on their behalf. These agents will ensure that for example the context is in advance adjusted and set before a user enters a certain place, or the device is turned-off while a user is attending a certain meeting. In that case, the user's profile will be an important part of the knowledge that embodies the agent. This profile would contain details on the user such as his level of expertise, his preferences, and his interests. The second level consists of associating devices with agents that will run on top of them. These agents will advertise to the agents of the user community the capabilities of the devices in terms of services to offer and needs to satisfy. Making agents runnable on resource-constrained devices is still not obvious. One of the most advanced platforms for that case is the Lightweight Extensible Agent Platform (JADE-LEAP) (see chapter 13); it is a Java-based FIPA compliant platform that allows a deployment of static agents on devices such as cell-phones (Adorni et al., 2001). The ability to have agents on small devices will become invaluable when building applications such as sensor networks that require large numbers of simple, inexpensive nodes.

Types of Agents. Usually, the identification of the relevant types of agents that populate an application is based on the roles that these agents will have to fulfill. In addition, the types of services that agents will have to offer to end-users or to peers of agents have an impact on this identification process. We believe that for an agent-based UC application, the following services could be considered and thus, should help in the selection of the relevant types of agents:

- Security services to be assigned to agents: users need a way to authenticate themselves before they undertake any operation. These operations depend on the authorizations given to users by a trusted authority.
- Scheduling and device broker services to be assigned to agents: in case an UC application involves a distributed set of fixed or mobile devices, scheduling the use of these devices to avoid conflicts is important. Extra scheduling is also needed if concurrent use of devices is doable. Besides the scheduling, the specification of the capacities of a device is an

initial step to complete. This specification is important for the broker services so, the matching between users' needs and available services can efficiently occur.

- Location data services to be assigned to agents: because of the limited storage capacity of the majority of mobile devices, the data that an UC application requires can be spread across the network and thus, need to be localized, fetched, updated, and restored back to its original host. Indeed, the networking infrastructure must provide robust data transfer with replication and discovery. Users must be able to count on their data arriving when they need and where they need to go without their direct intervention (i.e., explicit).
- Coordination services to be assigned to agents: an UC application may simultaneously engage several devices that have to interact despite the wireless communication channels. The coordination of these devices with respect to the process model underlying that application is important.
- Tracking and backup services to be assigned to agents: monitoring and logging the various operations are important in an UC application knowing that some devices could become down or temporarily unavailable. Thus, tracking services that trigger backup solutions are deemed appropriate.

Service and Agent Discovery. The current environment features the presence of multiple providers offering various services (e.g, weather forecast inquiry, stock markets tracking, vacation planning). To be able to invoke these services, future users have to discover and select the appropriate providers. Currently, the most common approach to connect users (or consumers) to providers and *vice-versa* is to insert an intermediate layer between them. Generally, specific types of components called brokers manage this layer. Brokers receive from the providers their advertisements of services and from the users their requests of services. Subsequently, the broker matches the advertisements to appropriate requests. Brokers are also similar to components known also as directory facilitator or agent management system (Ratsimor et al., 2002). Initially, an agent registers its services to one of these two components. Agents searching for services query either the directory facilitator or the agent management system to discover the agents that have the services. It should be noted that brokers, directory facilitators, or agent management systems are mostly assumed to reside in relatively powerful high-end static hosts and the communication links between agents are assumed to be wired and stable.

UC applications make the problem of discovering services more complex, since services and even agents can no longer be discovered using a static struc-

ture. Machines hosting directories are mobile and hence, can move out of the vicinity at any time. Moreover, mobile devices may be extremely resource-constrained. Different solutions are put forward for UC applications. In this chapter, three solutions are outlined. The first solution is based on agents and the use of the concept of peer-to-peer caching of services between nodes (Ratimor et al., 2002). The idea is that each participating device will be able to run a lightweight version of discovery service mechanisms. In addition, the device will be hosting at least an agent. Every node advertises its services to other nodes in its vicinity in accordance with certain policies. The second solution is based on a meeting infrastructure, on top of which agents, respectively acting on behalf of users and providers, meet in a common place (Maamar et al., 2001). Because agents can be enhanced with mobility mechanisms, they interact with their owner before they move to the meeting infrastructure. The idea of the meeting infrastructure is to offload most of the computing related to service discovery from resource-constrained hosts to fixed and reliable hosts. The third solution is based on the use of surrogates as Sun Microsystems promotes through its surrogate technology. This technology promotes the use of mobile objects. Mobile objects take care of orchestrating the interactions between clients and services. Objects are called surrogates and run on surrogate hosts.

Offloading the computing from resource-constrained hosts to fixed hosts has to meet certain requirements. These requirements have been discussed in (Messer et al., 2002) and summarized as follows:

- **Transparent, distributed execution:** it should be possible to execute an application on multiple machines without the application code being aware that multiple machines are being used. In addition, the platform should give the application the appearance of executing only on the client device.
- **Application partitioning:** it should be possible to dynamically divide the application at run time into two (or more) partitions that can be placed on different devices.
- **Adaptive offloading:** to be effective, it should be possible for the partitioning algorithm to consider available resources and execution patterns of the application. Based on either resource variation triggers or periodic re-evaluation, the platform should be able to adapt to load and execution changes to maintain a good partition decision.
- **Beneficial offloading:** the platform should only offload a portion of an application if doing so could benefit the user. It should be possible for the user to specify what is beneficial.

- Ad-hoc platform creation: it should be possible to create and tear down the distributed platform between a client and a surrogate at run-time. Clients should be able to determine which surrogate(s) are the most appropriate to be used based on factors such as latency of access and network bandwidth.

(Obreiter et al., 2003) state that today information is spread across large numbers of devices; many of them are at the same time wireless and mobile. To take advantage of all these devices considered as computing resources (sometimes could be exploited in clusters), their cooperation becomes a necessity. Besides the obvious cooperation that occurs at the application layer, device cooperation on lower layers (including link, network, transport, and discovery) may be necessary, too. Unfortunately, cooperative behavior means an increase of resource consumption which is not in the interest of mobile devices. To promote the importance of cooperation despite the issues of resource consumption and reliability of wireless networks, it is suggested to distribute incentives to the devices that display a cooperative attitude vs. those that display an uncooperative behavior (Obreiter et al., 2003). Various types of incentives have been put forward such as authorizations to use certain resources for a longer period of time, lowering charges of information routing, and enhancements of reputation.

In UC, agents have a role to play in promoting cooperation among devices and implementing incentives policies. We recall that agents can work as proxy for devices. Agents can be embedded into devices or run on fixed hosts from where they remotely control devices. Therefore, the attitude of devices will be depicted through their respective agent. Agents will for instance enforce the role of devices by supporting their commitments in cooperative interactions. Furthermore, the management of the different layers will be outsourced to agents, each agent being in charge of a specific layer. The interactions between agents will be of two types: horizontal and vertical. Horizontal interactions occur when two agents representing the same layer but from different networks will be cooperating, whereas vertical interactions occur when two agents representing different layers but from the same network will be cooperating. Besides to horizontal and vertical interactions, top-down and bottom-up interactions will exist between layers. Top-down occurs from the most abstract layer to the least abstract layer. And, bottom-up occurs from the least abstract layer to the most abstract layer.

6. Conclusion

In this chapter, we presented the field of ubiquitous Computing and the opportunities this field offers to the software agents research community. Ubiquitous computing will be a fertile source of challenges varying from technical

to social and legal aspects. This large variety of challenges calls for the collaboration of several communities besides the agents community. It is the belief of the authors that the advance of ubiquitous computing depends on overcoming the current limitations of mobile devices. Fortunately, these limitations are progressively being reduced as technology improves.

Today, a major source of distraction for users is due to the continuous need of managing their computing resources in each new environment (Sousa and Garlan, 2002). Situations of changes are multiple and call for a certain form of assistance that agents in general and adaptive ones (see chapter 8) can provision. Examples of changes are

- When users move to a new environment, their respective agent has to coordinate the migration of all the suspended operations to this new environment and negotiate the appropriate support to resume the operations.
- When the environment changes, agents have to seek to maintain the same quality-of-service for the components supporting users' operations. If this quality becomes incompatible, agents need to find out alternative configurations.
- When the context changes, agents have to adjust the operations that are affected by this change. Operations have requirements that depend on the execution context.

Ubiquitous computing applications are meant to make devices and their functionalities disappear into people' surroundings. People should be as little conscious as possible of the fact that they are using devices to get a certain work done. Ubiquitous computing does not promote a single device with which people interface. Rather, ubiquitous computing promotes multiple devices that adapt themselves to the person as well as the time and place of use. Agents are deemed appropriate to bridge the gap between people and (using) all these devices.

Chapter 20

AGENTS AND THE GRID

Service Discovery

Luc Moreau, Michael Luck, Simon Miles, Juri Papay, Keith Decker and Terry Payne

Abstract The Grid is a large-scale computer system, capable of coordinating resources that are not subject to centralised control, while using standard, open, general-purpose protocols and interfaces, and delivering non-trivial qualities of service. In this chapter, we argue that Grid applications very strongly suggest the use of agent-based computing, and we review key uses of agent technologies in Grids: *user agents*, able to customise and personalise data, *agent communication languages* offering a generic and portable communication medium, and *negotiation* allowing multiple distributed entities to reach service-level agreements. In the second part of the chapter, we focus on Grid service discovery, which we have identified as a prime candidate for the use of agent technologies: we show that Grid services need to be located via personalised, semantic-rich discovery processes, which must rely on arbitrary metadata about services that originates from both service providers and service users. We present **UDDI-M^T**, an extension to the standard UDDI service directory approach that supports the storage of such metadata via a tunnelling technique that ties the metadata store to the original UDDI directory. The outcome is a flexible service registry that is compatible with existing standards and also provides metadata-enhanced service discovery.

1. Introduction

The Grid (Foster and Kesselman, 1999), an open computing infrastructure that supports large-scale distributed scientific research and applications, has recently gained heightened and sustained interest from several communities. It provides a means of developing a variety of e-science applications including the study of genetic diseases (such as that described later in this chapter), particle physics making use of the Large Hadron Collider facility at CERN (see <http://eu-datagrid.web.cern.ch/eu-datagrid>), engineering design optimisation (Cox et al., 2001), and combinatorial chemistry. The underlying computing infrastructure also supports more general applications that in-

volve large-scale information handling, knowledge management and service provision (De Roure et al., 2003). Initially geared towards high performance computing, Grid computing is now being recognised as the future model for service-oriented environments, within and across enterprises, facilitating the formation of collections of coordinated services, or virtual organisations (Foster et al., 2001).

Large systems are naturally viewed in terms of the services they offer, and consequently in terms of the entities providing or consuming services. Grid applications typically consist of a set of such services that may be spread across a geographically distributed environment, and selected from a dynamically changing pool of available services. This service-oriented perspective, in which services (and their availability) may come and go, and collections of services are combined to achieve more complex tasks, very strongly suggests the use of agent-based computing (Luck et al., 2003). In this view, agents act on behalf of service providers, managing access to services and ensuring that contracts are fulfilled. They also act on behalf of service consumers, locating services, agreeing contracts, and receiving and presenting results. Agents are required to engage in interactions, to negotiate, and to make pro-active runtime decisions while responding to changing circumstances.

In this chapter, we discuss the issues of agent-based Grid computing in the context of the *myGrid* project, and then focus in more detail on the specific issues involved in service discovery, which we identify as a prime candidate for use of agent technologies. We begin by describing *myGrid*, which seeks to provide Grid middleware for bioinformatics (section 2), and then move to a general discussion of the role and use of agents in Grid computing for bioinformatics (section 3). Section 4 addresses the use of agents for one area of Grid computing in more detail, namely service discovery. We review the current technologies and introduce in section 5 a new service directory mechanism, *UDDI-M^T*, which augments the functionality of UDDI, the *de facto* standard directory for Web Services, with a metadata facility to better customise the publishing and discovering of services. As UDDI already offers a complex interface (for example, allowing searches on business categories and service names), *UDDI-M^T* uses a *tunnelling technique* for dispatching regular UDDI requests to a UDDI service, and intercepting *UDDI-M^T* metadata specific requests. Finally, we present conclusions and discuss further work.

2. The Grid and Bioinformatics

The Grid. The Grid is a large-scale computer system capable of coordinating resources that are not subject to centralised control, and which uses standard, open, general-purpose protocols and interfaces, delivering non-trivial qualities of service (Foster, 2002). As part of the endeavour to define the

Grid, a service-oriented approach has been adopted by which computational resources, storage resources, networks, programs and databases are all represented by services (Foster et al., 2002). In this context, a service is a network-enabled entity capable of encapsulating diverse implementations behind a common interface.

The service-oriented approach allows the composition of services into *workflows* in order to form more sophisticated services. Workflows are used for modelling the coordination between services, with each step in a workflow corresponding to an environment-dependent decision that must be made by some computational process.

In the e-business community, a service-oriented architecture is also being adopted in the form of Web Services, which have emerged as a set of open standards, defined by the World Wide Web consortium and OASIS, and ubiquitously supported by IT vendors and users. Web Services rely on the XML syntactic framework, SOAP for exchanging messages, the WSDL interface definition language (see <http://www.w3.org/TR/wsdl>), and the UDDI service directory (see <http://www.uddi.org>).

Against this background, the Grid community has extended Web Services to support resource management required by Grid computing. This effort has resulted in the Open Grid Service Architecture (OGSA), a Grid architecture standardised by the Global Grid Forum, that defines a *Grid Service* as a Web Service providing a set of well-defined interfaces, following specific conventions (Foster et al., 2002). In particular, Grid Services have some support for lifecycle management, and a conventional mechanism for discovery using service data elements (a service-specific type of metadata).

Bioinformatics. indexbioinformatics myGrid (<http://www.mygrid.org.uk>) is a pilot project funded by the UK e-science programme to develop Grid middleware in a biological sciences context (Moreau et al., 2003). To illustrate the functionality of Grid-based bioinformatics, myGrid has adopted an application that helps scientists study *Graves Disease*, a hormonal disorder caused by over-stimulation of the thyrotrophin receptor by thyroid-stimulating autoantibodies secreted by lymphocytes of the immune system. The Graves Disease application follows an *in-silico* experimental process typical of bioinformatics. In this process, the scientist: (i) attempts to discover information about candidate genes; (ii) makes an educated guess of the gene involved in the disease; and (iii) designs an experiment to be realised in the laboratory in order to validate the guess. This *in-silico* experiment operates over the Grid, in which resources are geographically distributed and managed by multiple institutions, and the necessary tools, services and workflows are discovered and invoked dynamically. It is a *data-intensive* Grid, in which the complexity is in

the data itself, the number of repositories and tools that need to be invoked in the computations, and the heterogeneity of the data, operations and tools.

In many resources, each record is analogous to an individual publication with not only raw data, but also additional annotations supplied by a small number of human experts (curators). Annotations are typically semi-structured text that may use keywords and controlled vocabularies, for parsing both by computers and by humans. Thus, in addition to a large number of data types, much of the valuable knowledge is locked into semi-structured text, under the premise that the scientist will read and interpret it.

In broad terms, myGrid follows common agent-oriented approaches in providing points at which automated processes can make decisions on what to do next depending on context. This manifests itself as automated service discovery, which we study in some detail in this chapter. First, however, we review the use of agent technologies in this context.

3. Agents in Bioinformatics Grids

Over the last few years, bioinformaticsindexbioinformatics has undergone a rapid and substantial change. The key problem faced in this domain is the multitude, heterogeneity and variability of data, tools and technical literature available to bioscientists. Although there are several well-known and highly regarded databases, they are not exhaustive, and new ones often appear with new and different data. Thus, any system intended for application to the bioinformatics domain should be able to cope with this dynamism and openness, and nothing addresses these concerns as comprehensively as the agent approach. Agents are flexible, autonomous components designed to satisfy overarching strategic goals, while at the same time being able to respond to the uncertainty inherent in the environment. On the one hand, agents provide an appropriate paradigm or abstraction for the design of scalable systems aimed at this kind of problem; on the other, the field of agent-based computing offers a set of technologies that may be used for particular purposes in certain aspects of the system, including personalisation, communication and negotiation. It is the latter aspect of agent technologies that we analyse in this context, and discuss below.

3.1 User Agent

The *user agent*, also known as a *personal agent* (Maes, 1994), is an agent in the sense that it *represents* a user within the myGrid system. It maintains a model of the user's goals and preferences, and uses these to make decisions and

ful feature, especially during workflow enactment, when a workflow is being executed and a choice of services becomes available. The choice should not be

made arbitrarily, but based on the priorities and circumstances of the particular user. For example, a user may have more trust in the accuracy of one service than in others. Instead of querying the user each time a particular service needs to be selected, the user agent can mediate the selection process based on prescribed preferences, or on prior experience. This adaptive behaviour is known as *personalisation*.

Another application of the user agent is as a contact point between services within the Grid and the user. By introducing an intermediary able to receive, for example, requests from services for the user to enter data or notifications about changes to remote databases, these messages can be delivered to the user only when the user is able and willing to receive them. Conversely, the user can delegate repetitive actions to the user agent, such as authenticating itself with a service before use.

3.2 Agent Communication Language

The idea of an *agent communication language* dates back to the DARPA Knowledge Sharing Effort, which led to the design of KQML, and was later followed by the FIFA Agent Communication Language (see <http://www.fipa.org>).

In MAS, it is common practice to separate intention from content in communicative acts, abstracting and classifying the former according to Searle's speech act theory (Searle, 1969). Thus, an agent's communications can be structured and classified according to a predefined set of message categorisations, usually referred to as *performatives*.

In seeking to integrate agent communication with standard Grid technologies, we have previously described how the idea of agent communication languages could be mapped onto the communication stack of Web Services. First, we focused only on the communication layer by encoding performatives and message contents in SOAP (Moreau, 2002). Second, we used the WSDL language for describing agents and the performatives they support (Avila-Rosas et al., 2002). The aim of this research was to expose agent capabilities as Web Services so that agents can publish their capabilities (and subsequently be discovered) in a UDDI registry. The approach turns out to be promising, because it offers a declarative communication semantics, which promotes interoperability, openness, dynamic discovery and reuse of agents. It also opens the agent world to the Web Services community, helping in the design of more complex interactions.

3.3 Negotiation Broker

Service users and service providers typically have different criteria regarding the quality and content of services, but can resolve the differences through

the use of negotiation. In Grid computing, one area in which negotiation can be particularly useful is in *notification* support. The providers of various services may want to send out notifications concerning improvements to tools, changes to databases or updates reflecting the state of enacted workflows, and so on. Other services or agents might want to register to receive a subset of these notifications. For stability, we consider asynchronous messages, and manage their distribution using a *notification service*.

The subjects (quantitative and qualitative) over which negotiation is undertaken could include the following forms of *quality of service*: the cost of receiving the notification; the topic (event category) of the notification; the frequency at which notifications are received; the generality of the change described by the notifications; and the format and accuracy of information contained in the notification message. These items, and many more, provide a metric for the quality of service.

An essential requirement for the smooth operation of any distributed system is that the consumer's demands of the service are met by the service providers. However, if these demands are not exactly met for some reason, the consumer may choose to *negotiate* with the publisher to find the next best quality of service that the publisher can provide. For example, the subscriber may require notifications weekly, whereas the publisher may only wish to provide them daily or fortnightly. In this case, the subscriber must choose between the available options or may decide not to subscribe at all, depending on their particular priorities. Alternatively, the publisher may be able to exceed the quality of service in several ways in which the subscriber may be unaware, and which could also lead to negotiation.

As the notification service must provide notification support for a potentially large and varying number of consumers, it should not change the contract covering the quality of service based solely on the results of negotiation between a single consumer and a publisher. Therefore, the notification service should have some control over the quality of service agreed upon. There are also other reasons why the notification service may usefully limit the interaction between the publisher and consumer, such as limiting one party's knowledge of the other for reasons of privacy or anonymity.

The *quality of service broker* described in (Lawley et al., 2003) is an agent conceptually contained within a notification service. This agent negotiates on behalf of each consumer wishing to receive notifications of a specified quality, and then produces a final proposal to both the consumer and the producer. It can negotiate with any of the publishers known to the notification service, and can also set boundaries on the agreed quality of service so that it is acceptable to the notification service.

4. Agent-Based Service Discovery for Grid Computing

Service discovery is a critical element in large scale, open distributed systems such as the Grid, as it facilitates the dynamic identification of resources abstracted as services. Providers may adopt various ways of describing their services, such as access policies or contract negotiation details. However, many resource consumers also impose their own selection policies on the services they prefer to use, such as derived quality of service, reputation metrics and. Consequently, both providers and consumers need to be able to locally manage and augment service descriptions with additional information, i.e., *metadata*.

The problem of service discovery is compounded by the plethora of different types of available service directories. Such services may include: *public* directories such as UDDI servers hosted by IBM or Microsoft; *specialised* directories such as the I3C bioinformatics service directory; *provider-specific* directories such as that of all the services hosted by a research institute; or even *local* directories such as the catalogue of all the services hosted by a laboratory for its own users. However, access to different types of service directory may require different protocols and query formats, with heterogeneous response formats.

Against this background, we have identified some key requirements that can enhance the process of service discovery by making the discovery process *personalised* to the user.

- 1 Users (not just service providers) should be able to attach metadata to service descriptions registered in service directories.
- 2 Users cannot be expected to systematically query all service directories in a discovery process. Instead, federating a selected set of service directories should provide a single point of access for the discovery process.
- 3 Users should be able to provide a semantic description of the task they want to locate, and the discovery should match the requirements against semantic descriptions of published services.

We will refer to the first two techniques as *syntactic*, whereas the third is *semantic*. Semantic descriptions can be used to specify what a service does in terms of the application domain (such as bioinformatics). Semantic techniques can then be applied to broaden or refine the list of services returned on discovery, based on the semantic descriptions and expert knowledge encoded in ontologies. For instance, queries for services of a general semantic type, such as sequence alignment tool, may also discover services described by a more specific type (sub-concept), such as BLAST tool. In the rest of this chapter, we shall only discuss the first technique, which allows users to attach metadata to published service instances. We refer the interested reader to (Lord et al., 2003) for a discussion of a semantic approach to service discovery.

4.1 Service Discovery Technologies

Service discovery has always played a crucial role in the evolution and deployment of distributed systems. Early distributed systems comprised collections of components (e.g., client-server or object-oriented) that were implicitly linked through function names, or linked through TCP/IP-based host and port addresses. The introduction of federated domain name servers (DNS) simplified and abstracted the use of numerical addresses by providing a registry-based mechanism for locating hosts. JINI (Arnold et al., 1999) used a similar approach as part of its Java-based distributed infrastructure. In this system, classes expose and publish their interfaces as *proxy objects* with the JINI discovery service. By searching for a given class-name, matching proxy objects can then be retrieved and invoked, which in turn call the remote service. While providing a mechanism by which services can easily be added, removed or replaced within a system, this approach is based on an assumption that there is a shared agreement about what a given service type is called (i.e., its class name) and that there was an agreed and well defined interface. Other distributed technologies support similar principles, including DCOM and Corba.

Web Services relax several assumptions of the JINI model. Unlike JINI, Web Services do not form well-defined class hierarchies, so it is difficult to locate services through class labels. To solve this problem, the UDDI service directory was introduced, to register both service and provider specific information. The UDDI registry can be searched through a list of service descriptions, but there is little support provided for searches based on the service's signature or user-defined data.

UDDI supports the *tModel* (Technical Model) construct, which essentially serves two purposes: it provides a namespace for a taxonomy, and a proxy for a technical specification that lives outside the registry. tModels represent a powerful but limited mechanism for augmenting service registrations with metadata; their expressiveness was demonstrated by encoding properties from the DAML-Services (DAML-S) ontology (Ankolekar et al., 2002) within UDDI records (Paolucci et al., 2002).

However, before tModels may be used, they need to be registered with a UDDI server and hence be unique. While this is suitable for mapping well defined specifications to tModels, it is inappropriate for specifying large numbers of locally used metadata attributes (such as a set of attributes that may be shared by a single organisation or domain). The UDDI V3 specification attempts to amend this oversight by defining a specific tModel, *general_keywords*, to allow simple unregistered key-value pairs to be attached to a UDDI entity. However, this solution is oriented towards metadata supplied by the service *providers*, not users, and allows only simple textual metadata as opposed to more complex structures. An alternative approach to storing explicit, personalised, and

possibly dynamic metadata that is associated with a service description is required to address these deficiencies.

In contrast to UDDI, the discovery services used by agent-based systems are typically designed to provide capability-based search, but provide little support for metadata-based search. These service registries index and search for registered services based on capability descriptions (or abstract descriptions of the service and its interface), rather than provider descriptions. Agents typically achieve their goals by identifying the types of services or tasks that need to be assembled together, and by delegating these tasks to those agents that provide the corresponding services. Such services are normally located by contacting one of several different *middle agents* (Decker et al., 1997). Indeed, some middle agents have been developed for the adoption of a semantically-rich capability description language such as LARKS (Sycara and Klusch, 2001) and DAML-S (Ankolekar et al., 2002) to facilitate the matching of semantically equivalent, interoperable services, despite the fact that labels or syntactic constructs in the returned interface definition may not exactly match those in the query.

Interestingly, however, many Grid projects require large numbers of service and domain specific metadata. This might include, for example:

- Perceived reputation of the service, which is critical to build a network of trusted services in an open environment;
- Perceived reliability of the service, which has more value if it is provided by a third party, and not by the service provider itself;
- perceived quality of service by taking into account external factors, such as network connectivity, bandwidth, latency etc.;
- Price for accessing a service (the user's institution may have negotiated a local price to access a resource, such as ACM or IEEE digital libraries); and
- Ontological descriptions of a service, which may differ if there are multiple ontologies or interpretations of a service. While we may imagine that a whole scientific community shares a common ontology, the very nature of undertaking research necessarily entails that ontology revisions will be created by those who undertake this research, and who will therefore want to use them in order to characterise services within their refined ontologies.

4.2 Notions of Services

One of the necessary elements to tackle in building a service directory relates to the differing notions of what a *service* actually is. Even within a uni-

fied model such as UDDI, the term is used in subtly different ways. In its most abstract form, the term *service* has two complementary meanings. For specifications driven mostly by the traditional distributed computing community (UDDI, WSDL), *service* tends to indicate a physical computing *entity* or entities that present some well-specified interface at particular physical endpoints. For specifications driven by the agents or more general AI community (DAML-S) (Ankolekar et al., 2002), *service* tends to indicate a *process* by which one may achieve a goal. These two viewpoints have significant overlap – an extremely common case in specification examples and in real implementations is one where a physical computing entity presents a well-specified interface which, in turn, enacts a process that achieves a goal. However, there are also situations that are harder to reconcile at this very high level of abstraction. First, from an agent perspective, a *service* could very well represent a large workflow quite explicitly spanning multiple physical computing entities (for example, composite DAML-S services), whereas a single service from the UDDI perspective may encompass many processes, each of which achieves different goals.

Not surprisingly, since the computational representation of semantic information has been a subject of study in AI for years, the most expressive models for semantic service description tend to be built around the agent-style service-as-process concept. While WSDL studiously avoids semantic information, UDDI does allow the assignment of predefined categorical values to both Business Services and Technical Services. Note that such categories are tied into the view of service-as-endpoint (e.g., geographical location) type of business, etc.

DAML-S attempts a full description of a service from the point of view that it is a process that can be enacted to achieve a goal. A full DAML-S service description incorporates three component perspectives: an abstract description of the service from the AI planning view (based on inputs, outputs, preconditions, and effects of a service – the service profile); the workflow view of the more primitive services needed to accomplish a complex goal (the service process); the mapping of the atomic parts of this workflow to their concrete WSDL descriptions (the service grounding). At its most complex, the DAML-S process view may be nested and include an explicit control model in order to monitor, alter, and possibly terminate the execution of a non-atomic service. Such models are analogous to emerging Web Service workflow proposals, such as WSFL (Leyman, 2001) and BPEL (Curbera et al., 2002b) and their associated standards, but this is beyond the scope of this chapter.

4.3 Service-Oriented Computing through Agent Technology

Service oriented computing, as described above in the context of the Grid, fits very well with agent technologies on the one hand, and the agent paradigm on the other. First, agent technologies of *middle agents* such as matchmakers and brokers can be used to address the problem of service discovery based on capability descriptions.

More generally, however, the paradigm of agent-based computing offers a way to view complex systems, with the area of AOSE providing ways of modelling and engineering such systems. These approaches make use of the *agent metaphor* to allow developers to reason about sophisticated behaviour in complex distributed systems, while avoiding explicit complexity in system design. In this view, the complexity arises out of the interactions of individual components at run-time rather than at design-time.

Clearly, however, distinct elements of the application must be identifiable as agents exhibiting flexible behaviour. Service directories matching the requirements given in the previous section are thus good candidates for an agent modelling approach for several reasons. First, they are federated, with annotations made in one registry being communicated to another with no guarantee of its inclusion in the latter: this is flexible social multi-agent behaviour. Such directories are *autonomous* in that they poll other registries according to some query, and *reactive* in that they may incorporate the results into their store, within the current environmental context provided both by a policy and by communication from other directories. Such flexible, autonomous, proactive, reactive behaviour demonstrates just those properties that characterise the agent approach.

Similarly, services surrounding service discovery can be viewed as agents. Agents can be present in the system as automatic publishers (or re-publishers) of services into multiple registries, as automated discoverers of services to be used in workflows, as personal agents adjusting service discovery to a user's preferences, and as automated executors that handle the invocation, composition and failure of services. Agents can also be used to regularly update metadata attached to a service description.

5. Architecture Design

The previous sections have identified Grid service discovery as a good candidate for deployment of agent technologies. Indeed, personalisation of content, flexible social behaviour through federation, and autonomy in the handling of requests, are all features of complex MAS. In this section, we describe a registry that can personalise content through a mechanism to attach metadata to service descriptions.

Since the types of personalised metadata that are required naturally vary greatly between individuals, organisations, and scientific user communities, an abstract and highly flexible representation is required. By regarding and implementing metadata items as triples that specify a relation between a subject and an object, arbitrary metadata can be described and queried via graph-based search criteria. This can be achieved through the use of RDF (the W3C Resource Description Framework, see <http://www.w3.org/RDF>), which underpins the Semantic Web effort (Berners-Lee et al., 2001).

While UDDI is the acknowledged standard directory service mechanism for Web Services, it is limited in the kind of metadata that can be stored about services, the ways in which it can be queried, and who can annotate service descriptions with metadata. Our previous work, UDDI-M, was an early attempt to associate metadata with services and maintain soft state information (Foster et al., 2002), based on *leases a la JINI*. Both ideas have been reused in a service directory with quality of service information (Shaikhali et al., 2003). With UDDI-M^T, we take a further step by regarding and implementing metadata as triples, which gives us access to Semantic Web technologies such as RDF, and powerful query languages such as RDQL over a uniform representation of information. UDDI-M^T works in conjunction with a UDDI service to provide precisely these extra capabilities, and eventually support for personalised directory service federation and semantic service discovery.

In this section, we describe the principles underlying the architecture of UDDI-M^T, which was underpinned by the following requirements during the design:

- 1 UDDI-M^T should be compliant with the UDDI specification and support future development in this direction.
- 2 Existing client applications and service providers should be able to make use of UDDI-M^T.

The key components of the architecture are depicted in Figure 20.1, where we see that UDDI-M^T is the point of contact for clients, used either for dispatching requests to UDDI or for processing them locally.

As far as implementation is concerned, UDDI-M^T was designed to be as generic as possible. First, all incoming requests are dispatched to the appropriate handler according to their type. Second, the UDDI-M^T backend is specified by an interface, which can be implemented in different ways: currently, We support the Jena triple Store (see <http://www.hp1.hp.com/semweb/jena.htm>) and relational databases.

This design assumes that all SOAP messages for the service directory issued by the client are routed to UDDI-M^T, which selectively filters them. This mechanism relies on the combination of the SOAP envelope and namespace contained in the message to dispatch the message to the appropriate handler,

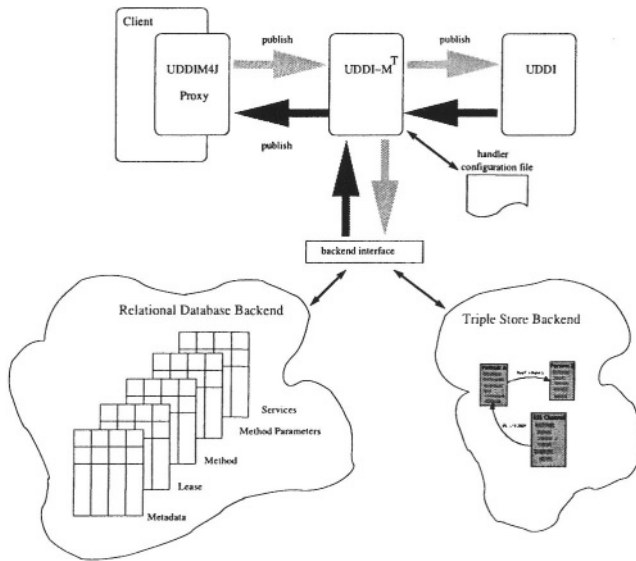


Figure 20.1. Architecture of UDDI-M^T

as specified by a configuration file. Messages with the UDDI namespace are directly *tunnelled* to other UDDI registries, whereas messages with a UDDI-M^T namespace are handled locally.

All metadata-related information is stored in the UDDI-M^T backend. Its interface is implemented in two different ways. On the one hand, we use a relational database with five tables for metadata, leases, methods, method parameters and services. On the other hand, the same information is encoded by triples in a Jena triple store, for which three implementations are possible: an in-memory store, a relational database or the Berkeley Triple stores (see <http://www.sleepycat.com>).

UDDI-M^T offers several extensions to UDDI. First, UDDI-M^T allows the association of metadata with services. Second, it supports a lease mechanism that requires services to renew their lease in order to maintain their registration in UDDI-M^T; such functionality is present in JINI (Arnold et al., 1999) and is also ubiquitous in the *Open Grid Services Architecture* (Foster et al., 2002). Third, UDDI-M^T is able to extract the information contained in WSDL files describing the interface. Fourth, UDDI-M^T extends the query mechanism of UDDI to allow searches of all the extra information it accumulates about services. Fifth, in the specific case of the Jena backend, UDDI-M^T allows users to express queries in the RDQL-query language (see <http://www.hp1.hp.com/semweb/jena.htm>), offering homogeneous ways of traversing the metadata graph associated with services.

While UDDI is defined as a Web Service, a programmatic interface UDDI4J (see <http://www.uddi4j.org>) is available for Java: it provides a client-side proxy with an API implementing the UDDI functionality, which allows programmers to abstract away from the messaging layer. We have extended this proxy, by subclassing it, with additional UDDI-M^T functions for managing leases and metadata. Thus, we preserve clients' binary compatibility. Indeed, a UDDI proxy can be transparently substituted for a UDDI-M^T proxy, in existing clients, since the latter is a subclass of the former. Clients do not have to use the functionality provided by UDDI-M^T , they can use the existing namespace specification and the calls will be directly tunnelled to the underlying UDDI service.

6. Performance Analysis

The Grid *checklist* (Foster, 2002) identifies “non-trivial qualities of service” as an essential feature of Grids. In taking this on board, the purpose of this section is to understand the impact of our design decisions on the performance of discovery. We focus our attention on two specific aspects. First, adopting the tunnelling technique reduces the implementation effort and allows us to maintain compatibility with evolving standards, but it comes at the price of SOAP-message forwarding; in the first part of this section, we analyse the cost of tunnelling. Second, the use of metadata in a service directory allows us to reduce the computational load on clients, while performing more selective and computationally intensive queries at the server side; in the second part of this section, we analyse the cost of metadata querying, and see how the use of the RDQL language, offering extended expressiveness to the user, impacts on the querying cost. A more detailed analysis can be found in (Miles et al., 2003).

Tunnelling Cost. Our hypothesis is that *the overhead introduced by the tunnelling technique is acceptable*. In order to evaluate such a hypothesis, we have set up the following experimental framework.

A UDDI-M^T service and its associated UDDI service are hosted in a Tomcat server. A client uses a UDDI4J proxy successively configured to use UDDI and UDDI-M^T . In order to avoid the cost of networking, both the client and services are run on a same machine, and communications take place through the “localhost” network device. We issue a UDDI-query that searches for a service with a specific name, for which a single instance has been registered. Figure 20.2 shows the overhead introduced by UDDI-M^T , which tunnels the request to UDDI. The tests were run on a Pentium 4, 1.5GHz, with 512Mb, using Tomcat 4.0 and the Registry Server 1.0_02, in the Java Web Services Developer Pack (1.0_01). The data plotted were averaged over 10 runs. The tunnelling overhead is 7.2%.

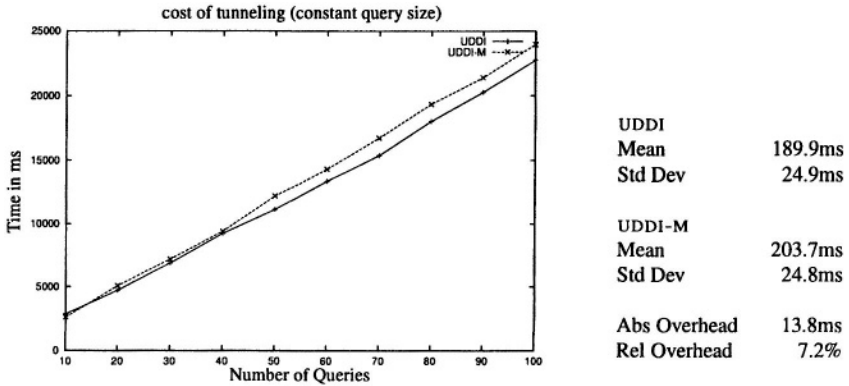


Figure 20.2. Overhead of tunnelling

We also evaluated the cost of tunnelling as the size of query results increases, but did not obtain any significant result, as the marginal tunnelling cost was noise compared to the querying cost.

Metadata Querying Cost. Our second hypothesis is that *using a triple store as an internal representation mechanism for UDDI-M^T is practical, and the use of the RDQL-query language can reduce communication costs, and offload the client, by performing some server-side computation.* For these experiments the associated UDDI is not involved, resulting in a commensurate reduction in query times from the previous experiments.

In Figure 20.3, we show the costs of attaching a property value pair to a service already registered, which we call a *property write* operation, and of finding a service with a given property value pair, which we call a *property read* operation. We used the two different backends, a MySQL relational database and Jena with the Berkeley triple store for these experiments. For the Jena backend, we used the Jena API to find the service with given metadata, and we did not rely on the RDQL-query language. We plotted the results in Figure 20.3a using a logarithmic scale to differentiate the curves better. *Our purpose here is not to compare persistent storage technologies, but to understand the cost of metadata management. We can see that read operations for both backends and the write operation in the Jena store are very similar.* We explain the higher cost for the write operation with the SQL database by the cost of storing information on disk, which is probably not measured with the triple store.

In Figure 20.3b, we used the RDQL-query language to search for a service satisfying 100 properties; 20 such services were found in the system. For convenience, we again plotted the Jena *read* line from Figure 20.3a. We can see that the RDQL-query engine processing a complex query that checks 100

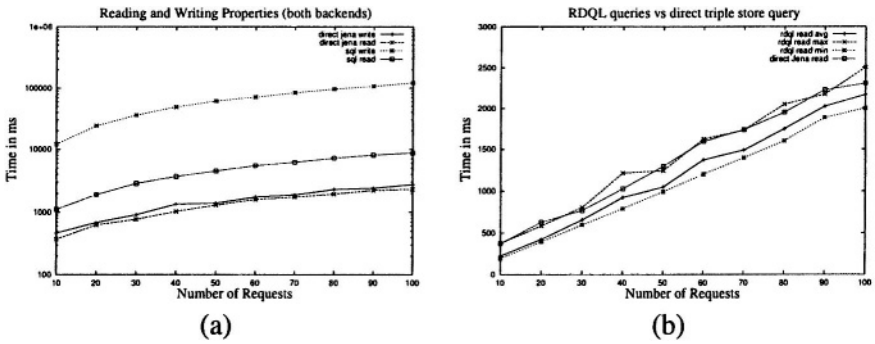


Figure 20.3. (a) Property read and write (b) RDQL Queries

properties marginally outperforms our direct use of the triple store API, which itself behaved well compared to a relational backend.

7. Related Work

The notion of agents has recently become popular in the Grid community. (Rana and Walker, 2000) advocate the use of the agent paradigm to integrate multiple information sources in problem-solving environments. (Busetta et al., 2001) describe a BDI agent architecture to simulate query optimisations in the Data Grid; their long term goal is to provide advanced and adaptable Grid services (of which query optimisation is one) based on agent technologies. (Rana and Moreau, 2000) review how agent techniques may be used to implement services at the computational Grid layer.

The CoABS (Control of Agent Based Systems) Grid (see <http://coabs.giobalinfotek.com>) integrates various heterogeneous agent-based systems, mobile agent systems, object-oriented systems and legacy systems. It is based on JINI (Oaks and Wong, 2000) for its lookup service and Java RMI for inter-agent communication, and tests of scalability of the registration mechanism have been undertaken in (Kahn and Cicalese, 2001).

Without mentioning agents explicitly, Furmento, Newhouse and Darlington (Furmento et al., 2001) discuss another JINI-based technique for federating resources. Their long-term goal is the building of a computational economy for the Grid. Several other projects investigate this idea of a computational economy, according to which an economics framework regulates the supply and demand of resources. In particular, Nimrod/G (Buyya et al., 2001) is a resource broker capable of *budget-based* scheduling, giving users incentives to trade execution time for economic cost.

From the agent side, the community has been very active in devising high-level interaction protocols able to coordinate the activities of suppliers and

consumers. Agents may *cooperate* in order to achieve a common goal, resulting in cooperative problem-solving which, sometimes, gives rise to adaptive behaviour. An alternative approach to this cooperation paradigm is the *market-based* model in which agents act as self-interested entities competing in a market, where goods such as computational resources are traded. Systems based on this paradigm have been shown to reach an overall equilibrium, in which resources are efficiently allocated (Clearwater, 1996; Kuwabara et al., 1996; Miller et al., 1996). The market-based approach gives good results in particular when resources become scarce, and is a specific case of the more general type of interaction among self-interested agents, *negotiation* (Jennings et al., 1998). As suggested earlier in this chapter, the key characteristics of negotiation are the presence of some form of conflict that must be resolved in a decentralised manner, by self-interested rational agents with incomplete information. Negotiation is the paradigm case of persuasion. It is a process by which agents come to a mutually acceptable agreement; apart from the work mentioned earlier, we are not aware of any of these techniques being applied in the context of the Grid.

Although the paradigm of agents has been used in the context of bioinformatics previously, this has not taken a Grid perspective yet. For instance, both (Decker et al., 2001) and (Bryson et al., 2002) used agent systems to federate data sources and tools in bioinformatics applications.

8. Conclusion and Future Work

In this chapter, we have discussed the roles and use of agents in Grid computing in general, but drilled down to explore service discovery in more detail. The examples of the use of agents that we have presented offer substantial new capabilities for Grid computing but still remain rather localised to some specific services. In particular, the full potential of agent technologies is yet to be exploited in future features of our service directory.

More specifically, the chapter describes the need for attaching metadata to services registered in service directories. This metadata describes functional and non-functional characteristics of services, and can be supplied by both publishers and consumers of a service. We have presented the architecture and the implementation of **UDDI-M^T**, an extension of UDDI, supporting metadata attachment and query. Our experimental evaluation has demonstrated the soundness of architectural design and implementation.

For the long term, agent-based computing also counts in its armoury a range of techniques for enabling individual components to collaborate with others, as well as for competing with others in the provision of services as may be found in bioinformatics. For example, the former aspects include issues in the construction of virtual organisations, whereby different services come together in

some coherent whole subsystem for a particular purpose; and issues in the regulation of open societies of services through the use of norms and electronic institutions. The latter aspects, for example, include the possible use of sophisticated auction mechanisms, or electronic marketplaces, for obtaining the best services or resources at the least cost to the user. Additionally, whenever interactions take place between different agents, the issues of provenance, trust and reputation become important. Though some work has been done in this area, the focus on both agent-based computing and Grid computing has been limited, with the majority adopting the stance of assuming complete trust, and avoiding the issue; questions of deception and fraud in communication and interaction, of assurance and reputation, and of risk and confidence, are particularly significant, especially where interactions take place with new partners.

Acknowledgement

This research is funded in part by EPSRC myGrid project (reference GR/R67743/01). Thanks to myGrid colleagues: Matthew Addis, Nedim Alpdemir, Andy Brass, Rich Cawley, Neil Davis, David De Roure, Vijay Dialani, Alvaro Fernandes, Justin Ferris, Rob Gaizauskas, Kevin Glover, Carole Goble, Mark Greenwood, Chris Greenhalgh, Yikun Guo, Simon Harper, Clare Jennings, Ananth Krishna, Peter Li, Xiaojian Liu, Phillip Lord, Darren Marvin, Karon Mee, Arijit Mukherjee, Tom Oinn, Steve Oliver, Savas Parastiditis, Norman Paton, Simon Pearce, Stephen Pettifer, Milena V. Radenkovic, Peter Rice, Angus Roberts, Alan Robinson, Tom Rodden, Martin Senger, Nick Sharman, Robert Stevens, Victor Tan, Brian Warboys, Paul Watson, Anil Wipat, Chris Wroe.

Chapter 21

ROADMAP OF AGENT-ORIENTED SOFTWARE ENGINEERING

The AgentLink Perspective

Zahia Guessoum, Massimo Cossentino and Juan Pavón

Abstract To promote the success of the agent technology the software engineering viewpoint should be rapidly addressed. This chapter analyses the existing approaches and discusses the future of agent technology from the software engineering viewpoint. It first highlights the properties of this new concept. It analyses then the existing methods and tools that have been introduced to facilitate the development of MAS. Finally, some promising applications areas are presented and a Roadmap for AOSE is introduced.

1. Introduction

Success of agent technology can be discussed from different perspectives. From the point of view of the software engineer, the agent paradigm will be accepted if it solves development problems (this means, improve the development process or allow the implementation of applications that otherwise would be difficult to build). Users, on their side, are only interested on services, and they do not care too much about the underlying technology, so agent technology would be of concern only if it allows the deployment of new services with some added-value (for instance, personalization). Agent technology, however, can facilitate requirements elicitation, which involves both users and developers, because the communication between them can improve, as agent concepts are, in principle, easier to understand by users than those common in the computer jargon.

But the final decision to invest in agent technology corresponds to managers. These can consider such investment if the new technology provides cost-efficient solutions and further business opportunities. Adoption of agent technology, as any novelty, implies some risks, as it requires changes in pro-

cesses and tools. At this point we meet again software engineers. They are the key for the change, so they should be able to evaluate, experiment, promote, and argue to convince their managers to invest. In this sense we are conscious that we have to provide substantial advantages of the approach to the software engineers community, and this will only happen if we show that it will clearly improve their activities, productivity, and creativity.

With this goal in mind, this chapter discusses the future use of agent technology from the software engineering viewpoint, addressing several issues: the agent paradigm as modeling technique, its associated analysis and design methods, supporting tools for development, validation and testing techniques, platforms for deployment of agent systems, and areas of application that can gain substantially from adopting an agent-based approach. It also describes the AgentLink Roadmap (Luck et al., 2003) for the adoption of agent technology, whilst co-existing with current practices, services and infrastructures. As it has been described in the rest of the book, there is already experience in developing agent-based systems, and there is also some effort in defining methodologies for building software using the agent paradigm. After around a decade of these experiences, the question now is whether software developers can adopt the agent approach for software development and how to integrate this with current practices (e.g., object-oriented and component based software). This has associated many concrete questions that we address in the following sections: Are there needs that current practices do not fully satisfy? How can agent-based solutions improve the software development process? Are there standards? Are there working systems of agents in the net? What do they do? Where can be found those agents? Can we buy software agents? How much do agent-based systems cost (in terms of deployment, integration, tools, learning, etc.)? Some answers are described in previous chapters of this book. Here we integrate some of these results to present a vision of what the future of agent-based computing and specifically of AOSE will be.

This chapter organizes the discussion as follows. Firstly, in section 2 we overview those features that make the agent concept interesting for modeling complex systems, and in this perspective it is possible to consider it as an evolution of object and component based approaches. Given this, in section 3 we consider how this is applied along the development lifecycle, from a methodological perspective. In concrete, we are considering the FIPA proposal for the future in MAS design. Section 4 follows with a description of significant tools for implementation, deployment and execution of agents. Tools, in fact, determine the maturity of the technology, so they can provide a good picture of the evolution and degree of adoption of the agent paradigm. The opportunities for using agent technology are the subject of section 5, where some promising application areas are reviewed as candidates for making profit of this technology. Section 6 takes as reference the AgentLink Roadmap for agent-based comput-

ing and describes a roadmap for AOSE. Finally, the conclusions point out the risks and opportunities for agent technology success, which relies on the new ideas for AOSE and the role of standards.

2. Agents as a New Modeling Paradigm

Agent related concepts provide new ways to model complex and dynamic systems. As it is discussed in (Zambonelli and Parunak, 2002), today's software systems are getting higher degrees of complexity in different respects, not only in size, as other factors are combining together, for which the agent paradigm provides some solutions:

- The environment of the software systems is more and more dynamic, subject to continuous changes. Different (not necessarily software) systems co-exist in the same environment, either collaborating or competing. Their actions have an impact on the environment, and this happens concurrently. Therefore, it is not possible to assure that an action will have the expected result. In this sense, goal-driven approach for system design is quite convenient, as the goal is more stable entity than others that may be used to define the system state. Also, as there are several ways to achieve one goal, the system can be conceived to adapt to changing conditions by adopting new action courses.
- There are more and more computing devices everywhere, with different capabilities, and connected to (mostly wireless) networks. This implies higher degrees of distribution, in the management of the system entities, in the location of control, and in the interactions. The agent approach assumes these considerations in its foundation. Agents are conceived as autonomous entities which can reside in a node of a network, and even migrate from one to another in the course of their lives.
- Knowledge processing and management. There is an increasing need to reason about something more than raw data, to provide knowledge-based services. At this point, new mechanisms for information processing are required, and interaction among system components requires higher level of abstraction, with more support for semantic processing. In this sense the use of ontologies and agent communication languages suppose a step forward with respect to traditional object-oriented computing.
- Usability of computer-based systems has increased as far as more people use these. Higher degrees of personalization become an important factor for service acceptance and differentiation. This implies highly reconfigurable systems, where there is a need for special processing for each user with specific data. This is often addressed by considering one agent

as personal assistant for each user, with capabilities to learn and adapt to the changing user's profile.

In spite of these considerations, the agent concept should not be seen as a radical new paradigm but as an integrating paradigm, or an evolution of current distributed object systems (in fact, the border between agents and distributed objects is quite fuzzy, as many papers in agent conferences show). Traditional distributed object computing has put emphasis on the features of middleware and associated services, and adopts current object-oriented methods and tools. Agents appear to cope with the issues above, when computing gets ubiquitous and intelligence appears elsewhere in a diverse range of devices, from classical servers to ambience computing.

Agent technologies are founded on distribution technologies and object orientation, and integrate them with artificial intelligence techniques. From distributed object computing takes the autonomy of agents, which can be distributed (therefore supporting system scalability) and interact through message passing. To make distribution feasible, support services are defined, such as white and yellow pages, in a similar way as in current state of the art middleware.

From object-oriented modeling, analysis and design methods are extended to include new ways of reasoning about system conception. When thinking about MAS, responsibilities are clearly separated from one agent to other, and these are characterized in terms of goals rather than as a set of functions. This is important in the sense that goals are considered as more invariant than input-output relationships (functional approach) along system life-cycle evolution. And this is one of the points where contributions from artificial intelligence field come into place, for the modelling of agents and agent communities behavior. Given that the agent is a goal-based entity, its behavior can be conceived as a reasoning system, where decisions on which task to execute at a given moment depend on current knowledge of the environment, the status of achievement of goals, and actual capabilities of the agent (and surrounding agents).

With this respect, agents are said to work at the knowledge level, and follow the rationality principle (Newell, 1982). This has the advantage of providing a high degree of flexibility at individual level (each agent). Also at organizational level as interactions among agents are defined with intentions (an agent expects some action from other agent when interacting through a given primitive), and with semantic processing (agents understand ontologies, which give semantic meaning to the words used in their messages; however, how each agent process each message it is up to the agent).

At the end, the agent concept, from an engineering viewpoint, can be considered also as an extension of the object-oriented component model. Agents can be deployed in a distributed system fairly easily. And can be configured,

not only in some parameters, but in behavior. To an extreme, agents can learn new procedures, and even new interaction languages and protocols. A MAS then reflects a set of highly configurable entities. But also the MAS, as an organization, can be reused. New systems can be conceived as the combination of agent organizations, each one providing services and relying on services of other organizations. With this respect, the agent paradigm provides for both horizontal and vertical decomposition of complex system development. Because of the growing possibilities of such approach, work on coordination of agent systems is considered as fundamental.

3. Methods for Building Multiagent Systems

Building MAS is a complex activity that takes both advantage and complication from the same nature of agents. In fact, we should note that while many modern MAS are implemented with object-oriented languages (and therefore need to be specified down to this level), they want to reflect the social solution to a problem that has not been tackled with an object-oriented approach but with very different abstractions (communications instead of method invocations, freewill collaborations instead of client-server servitude). The agent paradigm could serve as a tool to decompose the problem complexity and easily manage very large systems.

In such a context, several researchers have tried different approaches to systems development. These works usually reflect the situation that originated them; we have methodologies arising from specific needs (e.g., robotics), a strong background in a discipline (usually artificial intelligence or software engineering) or the exploration of a specific paradigm (adaptive or holonic agents). We can consider the differences in these origins as a richness, the overall scenery is huge, variegated and full of interesting perspectives.

Another important factor in this context regards the boundary of the system. By now the greatest part of the applications deals with closed systems and these, also because of security concerns are, and probably for some time, will be a must in commercial and industrial applications. This situation cannot though be too lasting. Agents are part of societies and an important step in all the societies growth towards a full maturity consists in the openness. This will bring a greater attention for the related problems in all phases of the development.

The AgentLink Roadmap (see section 6) divides the past, present and (a possible) future in the development of MAS into four different phases. Up to recently we approached these problems looking for ad hoc solutions, now we are going to benefit of more general development methodologies. To go beyond this phase we have to proceed towards the adoption of well established standards that include a consistent support for patterns. This standardization will

encourage the production of a new generation of design support tools that will increase the dimension of manageable systems and the designers/programmers productivity. In this sense, we can expect the same process that happened with object-oriented methods that integrated around the UML standards.

The increasing dimension of MAS is also driving a change in what it means to design, implement, deploy, test and maintain such systems (Zambonelli and Parunak, 2002). Probably, in the next future, we will not design complete applications but rather add new functionality by adding one or more agents to enormous existing systems. This also mean that implementation choices will be strongly conditioned by the operating environments (existing systems) and even more by the respect of well established standards. In fact, the idea of deploying a whole new system is not valid anymore. Systems will exist (in the network) and will just evolve by adding new agents, evolving the behavior of existing agents, or firing old-fashion or unused agents. Once deployed the new agents will face an open society where unsuspected threats could arise and crucial elaboration nodes could fall; the system in its entirety should be able to survive and achieve its goals also if some of the agents will not. In this situation, testing system validation is different from the actual one. We will be more concerned with the overall (and emerging) behavior of the society rather than the performance of the single agent that could even fail in doing its duty if some solution will come from the remaining part of the MAS. Maintenance, at the end, will be more concerned with updating existing systems on the fly (that means substituting some agents/adding new ones with new features) rather than stop and replace them with entirely new solutions.

The risk involved in the quick and interesting growth that we can observe in the agent community, is that the great speed of advancement could bring all the involved researchers and practitioners to forget that we should not re-invent the wheel. The problem of designing a system has been discussed since a long time (first “modern” methods, like DeMarco’s Structured Analysis, belong to the 1970s) and this important heritage, it is sometimes forgotten by agent people.

In this phase of the MAS development it seems that some of the aspects of the whole process are less deepened than others. This is the case of the organization that is behind the software production and the maintenance concerns.

Most attentions in this period are directed to technologies, procedures and artefacts, and mainly to the so called design methods, “*a structured approach to software development whose aim is to facilitate the production of high-quality software in a cost-effective way*” (Sommerville, 2001).

Several methodologies for designing MAS exist in literature, and this book reports Gaia, Tropos, MaSE as examples of generic methods and ADELFE, MESSAGE, Prometheus and SADDE as specific-purposes approaches. Many other diffused methodologies exist, e.g., ADEPT (Jennings et al., 2000), MAS-

SIVE (Lind, 2001), PASSI (Cossentino and Potts, 2002), for specific problems (like ADEPT devoted to business process management) or not.

This abundance reflects the different needs and approaches of different designers and it is reasonable to say that an unique specific methodology cannot be general enough to be useful to everyone without the possibility of some level of personalization.

3.1 A FIPA Proposal for the Future in MAS Design

In its roadmap, the AgentLink community indicates the identification of some standard in design methodologies as one of the milestones of the path towards the success of agent-based systems.

The FIPA answer to stimuli like this consists in the constitution of two specific technical committees (TC); the first deals with the identification of a new unifying approach to the design of MAS and the creation of the consequent standard proposal (Methodology TC); the second one (Modeling TC) aims at defining a modeling language (Agent UML) that starts from the experience of the Unified Modeling Language and extends it in order to model MAS.

A fundamental step towards the maturity in design processes for MAS has already been done with existing methodologies and with the intent to take profit by this, the Methodology TC will adopt the paradigm of method engineering (Saeki, 1994). According to this approach, the development methodology is composed by a method engineer assembling pieces of the process (method fragments) from a repository of methods built up taking pieces from existing methodologies (ADELFE, AOR, Gaia, INGENIAS, MESSAGE, PASSI, Tropos, etc.). Obviously if necessary fragments are not available he/she could create the ones he/she needs. The result will be the best process for the designers (that will actually perform the design) specific needs.

Method fragments are composed of essentially three elements: the process to be followed to achieve the fragment objective, the artefacts to be produced, and the roles played by the involved people. The OMG SPEM (OMG, 2002) standard could be largely applied to the description of the process aspects of the method fragment and in fact, it is currently under evaluation with very encouraging partial results. The artifacts to be produced depend on different aspects: what is to be designed (and this relies on the MAS meta-model) and how the designer will describe his/hers choices (artifact notation). This last topic is the specific work of another FIPA Technical Committee (the Modeling TC) and a standard FIPA modeling language will be standardized for these purposes. Globally we can see that a complete standardization strategy has been drawn that should give in the next few years a good support to the MAS development.

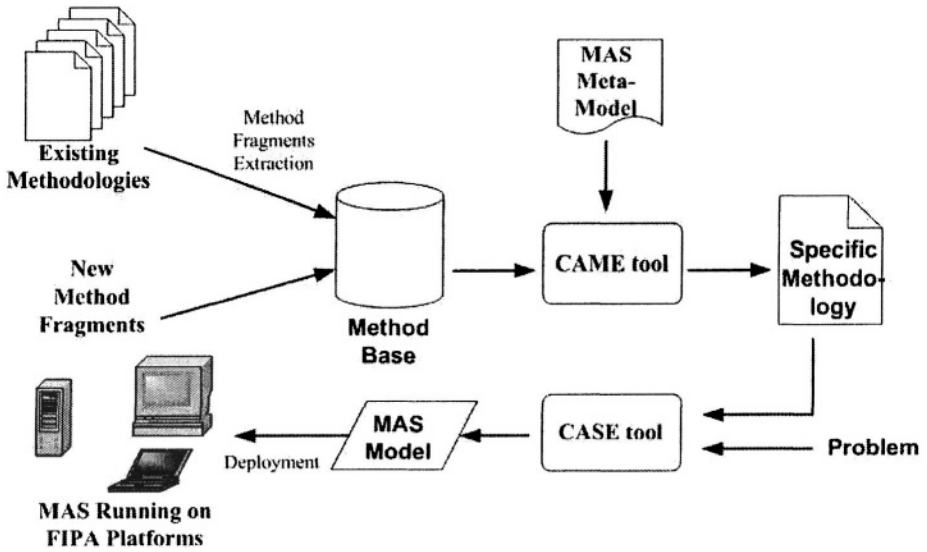


Figure 21.1. The method engineering process for MAS design currently proposed by FIPA

Looking now at the complete method engineering process we can see that during a real design process (Figure 21.1), the designer (or better the method engineer), before building his/hers own methodology, has to select the elements that compose the meta-model of the MAS he/she will build.

This operation will be supported by a CAME (Computer Aided Method Engineering) tool that offers a specific support for the composition of a methodology from existing fragments or with new ones.

Once the methodology is composed, the designer or the design team could perform the established process (supported by a specifically generated CASE tool) obtaining a model of the system that solves the faced problem. Finally the agents could be deployed on the required platforms obtaining the running MAS.

In the last years, the method engineering approach proved successful in developing object-oriented information systems (Tolvanen, 1998). We should evaluate the importance that this approach had in the object-oriented context considering not only its direct influence (not so much companies and individuals work in this specific way) but an indirect consequence of it: the most recent and diffused development processes (for example RUP, the Rational Unified Process) are not rigid but they are a kind of framework within which the single designer can choose his/hers own path.

The introduction of the method engineering paradigm in the AOSE has a peculiar problem. While in the object-oriented context the construction of

method fragments (pieces of methodology), the assembling of the methodology with them and the execution of the design rely on a common denominator (the universally accepted concept of object and related model of the object oriented system), it is not so for MAS. It is a matter of fact that, there is not an universally accepted definition of agent nor it exists any very accepted model of the MAS.

We could describe the system (object or agent-oriented) design process as the instantiation of the system meta-model that the designer has in his/hers mind in order to fulfill some specific problem requirements. This meta-model is the critical element in applying the method engineering paradigm to the agents world. It is a structural representation of the elements (agent, role, behavior, ontology, etc.) that compose the actual system with their composing relationships; this includes generic elements (e.g., the agent) but also specific ones (e.g., the cooperative agent referred in the ADELFE methodology) and its absence could be observed in the different uses that different authors make of these concepts, for example the behavior, that are often presented with slightly different meanings, granularity or abstraction levels. The availability of a standard definition of the MAS structure becomes therefore a strategic issue for the success of a MAS development process that wants to be largely applied and diffused.

3.2 Validating and Testing MAS

Today no one can claim that requirements for a software system are well-known and stable. This situation is even more appropriate with MAS, given the kinds of problems they address. Evolution in requirements brings the necessity of changing the software in a continuous series of iterations that add new parts and change existing ones, increasing the risk of introducing defects, deteriorate performance and stress the adopted technological solutions sometimes beyond their limits. The resulting quality is not so high as it was expected and the customer could be a little disappointed with the current release. A common solution is to propose an evolutive patch and, obviously, the described process will happen again. In the last nightmare scenario, we intentionally neglected the fundamental role of a great part of the software engineering research whose activity spreads over the well known phases of debugging, testing and verification. These are the keys of a successful software but the advances in these fields are undoubtedly not sufficient. In the following we will examine the main factors that will characterize the progress in this field.

Debugging. It often happens during the coding activity and consists in analyzing the given program and extending/changing its behavior in order to get a correct behavior and meet the specifications. Considering that often agent-based software is implemented with object oriented languages, in combination

with other techniques (for instance, rule-based systems), and in a distributed environment, debugging a MAS is not a simple task at all. Basically developers rely on traditional object-oriented debug tools and, with the exception of some professional environments, support for other paradigms is scarce. Also, some FIPA-compliant platforms provide monitoring for messages among agents, and this is basically all the support we can get. In summary, there is clearly a need for tools that integrate the different programming paradigms and able to monitor the execution and communication of agents. In developing a MAS this is even more useful considering the additional complexity introduced by the agent nature of the system; interaction of all the tools involved in the design/coding/testing chain gives the opportunity of tightly combine the agent-level specification with its translation in the coding language and finally modify it.

Verification and Validation. Verification (answering to the question “*Are we building the product right?*”) and validation (answering to the question “*Are we building the right product?*”) of MAS have been discussed in several works with the use of different kinds of formal specifications. The real limit of this approach is that it is complex and time-consuming; this is often in contrast with the market rules and its applications are confined in only some specific applications. In the future we will, hopefully, have an automatic support for this activity where more intelligent tools will be able to verify the respect of some specifications even if they are not provided in a exclusive formal way. We have to admit that this still remains a very open research field.

Testing. Not so much research efforts have been spent in testing MAS. Topics to be explored regard the identification of test cases (possibly with the support of specific tools) and the creation and tuning of new techniques for testing agents. While test cases identification and planning can be seen as more related to requirements than the implementation and therefore not so much influenced by the MAS nature, totally different considerations can be done for testing techniques. Often we talk about unit testing and integration testing addressing the difference scope of the two activities related to the single unit (agent in MAS testing) rather than its integration with the remaining part of the system. Working with agents, these definitions have a slight different meaning. Agents are highly encapsulated entities and adding new features in a system, often involves introducing new agents rather than changing the existing ones. Testing the agent behavior (unit testing) is much more complicated than testing an object-oriented sub-system since often agents are not deterministic. Classical techniques like equivalence testing (based on the assumption that the unit behavior is the same in a range of input values) are almost useless with purposeful agents whose behavior is not triggered only by external stimuli but also

by a specific (and changing) will. Integration testing is again different in agents from objects because of the different nature of entities relationships. Objects essentially relate by strict method invocations while agents interact with communications that have several freedom degrees (the same agent can participate in conversations using different languages, ontologies and rules without losing the meaning of the act). Integration is not only concerned with entities interfacing but it also looks at the resulting collective behavior. Researchers and practitioners are still exploring different ways of coordinate agents in order to obtain a specific behavior and the results although very interesting are sometimes not definitive and this, partially, justifies the limits that we have in the subsequent testing of these systems.

4. Tools for the Implementation, Deployment and Execution

The distributed nature of MAS, and the integration of different paradigms for building this kind of systems, demand the adaptation of current state-of-the-art methods and tools to the specific characteristics of MAS:

- **Openness.** New agents can be dynamically added and existing ones can disappear.
- **Heterogeneity.** It is an important property of complex systems which can be often modelled by MAS. Heterogeneity requires a high level of interoperability between heterogeneous agents. Several kinds of heterogeneity are considered such as multiples implementation languages, multiple execution platforms and multiple knowledge representation.
- **Distribution.** MAS are inherently distributed. They have therefore the advantages of distributed systems, but also the design, deployment and execution difficulties. The agent paradigm is devoted to the development of complex systems. The latter are often very dynamic, adaptive and large scale and requires reliability, security, interoperability and scalability. However, existing distributed systems solutions are often applied statically by the programmer before the application starts and do not deal with scalability.

This section discusses agent tools and platforms which have been proposed to implement, deploy and execute MAS.

4.1 Tools for Designing Agents

Some decades ago architects were used to design buildings by hands or just using very simple calculation tools. Nowadays, it is sure that no one modern architect will accept to design even a two floor building without the support of a

computer and some software (architectural design and structural dimensioning programs). All of us are aware that software is not less complex than a building though only in the very last years the use of tools for supporting the design phase has become widely spread.

The application fields of these tools vary from requirements elicitation to design, testing, validation, version control, configuration management and reverse engineering. The different phases of the software life-cycle can be covered using separate tools (sometimes with some level of inter-operability) or an unique environment (an integrated collection of tools) that is often process-oriented.

The main requirements that a tool should offer, in order to support the future needs of the agent designer, are:

- Usability. MAS-related concepts are more difficult to study and manage than the classical object-oriented ones (mainly because a MAS involves more concepts than an object-oriented system), moreover designers get often skilled with objects before than agents and therefore they receive some kind of imprinting from their initial field of knowledge. This should guide agent-tools developers to produce applications that in guiding the newbie into this new world do not neglect the proper attention for his/hers background.
- Multi-view support. The design of an agent-oriented system involves preparing several different representations of it, each one addressing a different level of abstraction or point of view on the software. An important role can play in this direction the AUMML proposal (see chapter 12) that aims at defining a complete language for agents modeling that starts from the widely accepted UML and introduces MAS-specific notational elements.
- Traceability. Different views of the system should represent the same unique software and the designer needs some help in order to coordinate the different artifacts he/she produces. Going through the different stages of the development process, it is easy to interrupt the correct, logical flow of the design refinement for example forgetting the specification of an element or introducing inconsistencies. The problem is particularly important in MAS since they introduce new concepts, abstractions and logical steps that complicate the design process. Tools can be very helpful in achieving traceability. They can provide automatic checks of many different aspects and they are not so influenced by the system complexity if its (expected) underlining structure (the meta-model) is clear and the design process is completely defined.

- Specific support for the software process. Beside the conventional needs presented by several projects (e.g., information and legacy level systems), there is now a consistent part of software that is strongly effected by time-to-market constraints. In many fields (e.g., e-commerce), once a customer need is detected the gap before the introduction of the piece of software tackling with it, is usually a strategic period for the involved company. A competitor could be ahead of time and occupy the new market slot. Quality of the first release is, in this case, not the primary goal of the development team. A limited but working program is always better, in this scenario, than no software at all. Agile design methodologies (that are becoming quite diffused in other contexts) are still not present in the MAS development scenery but it is likely that their need will be sharply perceived in the very next future. We already discussed the future of MAS development process in the previous sections and again we would like to underline the concept that the agent society has to overcome the actual experimental phase in which many systems are developed with a low attention for rigorous (or sometimes ad-hoc conceived) design methods and go towards a full maturity stage where industrial quality programs are released after performing problem (or context) specific life-cycles.
- Generality. Let us suppose that a large software house decides to move into the agent world and to produce only agent-based software. At the first step some developing tools and language will be identified and workers will be trained to work with them. This is a costly phase and the company management will be very careful about the outcome of this effort. In this scenario it is not presumable that the chosen environments could be specific for only some contexts. Applications produced by such a company will probably vary with time and address very different concerns; supposing that specific tools will be adopted (and studied) for each different project it is not realistic. We need therefore to look at an highest level of generality for tools we will produce in the future. While there (still) will be some space for domain-specific solutions (for example robotics or telecommunications), more often, general purpose yet configurable environments will satisfy the real needs in many cases.
- Tools integration. We can easily forecast a scenario in which the designer can choose his/hers tools from a consistent range of possible alternatives. Different programs will be used to support the requirements elicitation, actual design and final implementation coding activities. There is an undoubted technological problem in making all of these tools to cooperate in a plain way. The solution in the object oriented world comes from choosing an easy, standard but enough struc-

tured inter-operation language such as XML (or its derivative like XMI and some others). This will be probably the initial choice in the future agent design environments but we think this forgets an important aspect of MAS: they are strongly ontology-based. While XML can be considered a good vehicle for information it does not provide (by itself) the proper structural support for each specific exchange operation. It is more likely that knowing the ontology of the domain where the agent system will be deployed also the tools involved in its design can adapt to it and interact using ontology-based communications that deal with specific problem abstractions rather than with pre-configured structures.

4.2 Agent Implementation Tools

To make concrete the various research in MAS and facilitate the implementation of applications, several agent implementation tools have been proposed. In the present state of research and development, we find contributions on agent architectures and on agent implementation languages.

Agent Implementation Languages. Several languages have been introduced to facilitate the implementation of agent societies. The most common approach is based on the provision of libraries for common-use programming languages, such as Java, which are enriched with utilities for agent communication and services (e.g., FIPA based) and the use of other programming approaches (declarative, rule-based, etc.). For instance, several MAS have been implemented with actors (or active objects) languages which are extensions of object oriented languages (Gasser and Briot, 1992).

Another approach consists on the definition of a brand new language, as in the Agent Oriented Programming (AOP) work (Shoham, 1991). AOP is a new programming paradigm that supports a societal view of computation. In AOP, agents (an agent is defined by Shoham as “*an entity whose state is viewed as consisting of mental components such as beliefs, capabilities, choices, and commitments*”) interact to achieve individual goals. The agent behavior is described by a rule base that reacts to received messages and changes of agent state. The agent dynamic is therefore implemented by a first-order forward chaining inference engine.

A substantial amount of work has been done in pursuit of a complete formalism to develop the AOP idea, see, e.g., PLACA (Thomas, 1993), and Concurrent-METATEM (Fisher, 1994). Few attempts, however, have been made towards developing an actual, useful agent-oriented language. The result is that the few actual languages in existence are far from achieving the promise of AOP and are of little practical use. The development of a useful agent-oriented language should rely on existing agent and multiagent architectures. The latter functionality provides the basic primitives to facilitate agent imple-

mentation and their interactions, see the languages Claim (Fallah-Seghrouchni and Suna, 2003) and 3APL (Dastani et al., 2003).

Agent Architectures. Several agent architectures have been proposed, Two main approaches can be distinguished: cognitive and reactive, a survey is given in (Wooldridge and Jennings, 1995b) and examples are given in (Avouris and Gasser, 1992). In the cognitive approach, each agent contains a symbolic model of the outside world, about which it develops plans and makes decisions in the traditional (symbolic) Artificial Intelligence way. On the other hand, in the reactive approach, simple-minded agents react rapidly to asynchronous events without using complex reasoning. Neither a completely reactive nor a completely cognitive approach is suitable for building complete solutions for real-life applications. Hybrid models (Ferguson, 1992; Muller and Pischel, 1994) have been proposed to combine the advantages of both reactive and cognitive models. In these models, agents are decomposed in a set of modules which can in turn be of a reactive or cognitive nature. However, the problem with such models is that of implementing various types (reactive, cognitive) of agents. Indeed, real-life applications require often various types of agents and variable granularity. For instance, these hybrid models cannot be used to implement small agents such as ants.

A good architecture may be seen as an open model. This solution is provided by modular architectures which are based on software components. A modular agent architecture makes the agent an open system. Modularity introduces flexibility and allows to change easily components with the aim of improvement or tests. Modularity provides thus several advantages: *(i)* possibility to have variable granularity of agents; *(ii)* possibility to have agents with adaptive structure, each agent can dynamically change its components and the relations between these various components; *(iii)* possibility to integrate different agent models; and *(iv)* possibility to include a library of reusable components. A good agent implementation tool should be based on a modular agent architecture and provide libraries of components. On this way, each agent has one or more components, e.g., communication, interaction protocols, etc. This approach facilitates the reuse and integration of existing paradigms, e.g., production rules, and state machines. An example of modular architecture is given by (Guessoum and Briot, 1999).

Agent architectures provide several facilities to build MAS. To develop a MAS, one has to know all the components or classes of the library (agent classes, simulation classes). These development difficulties raise from the diversity and complexity of agent and multiagent concepts (coordination, interaction, organization, etc.). This complexity makes the use of most existing agent tools very difficult to non-owners (developers) of the tools. To deal with this complexity, PTK (see section 4 for more detail) proposed to provide tools

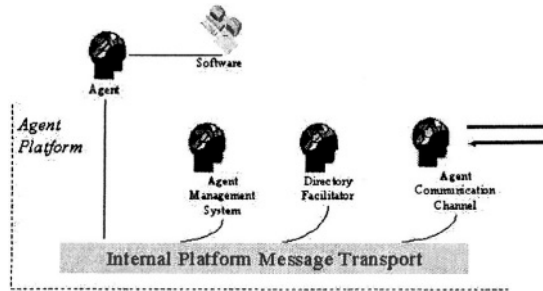


Figure 21.2. Overview of the FIPA architecture

to facilitate the specification of MAS and to elaborate a process development. Another way to facilitate this choice is to make abstraction of some technical details and define meta-models by using the MDA (Model Driven Architecture) approach introduced by the OMG (OMG, 2001).

4.3 Agent Deployment Tools

The first MAS (Avouris and Gasser, 1992) were composed of a set of homogeneous agents which run on one computer or a local network of computers. So, the deployment problem was kept off. However, Recent real-life applications (see section 4) are often open and distributed at large scale and must run continuously without any interruption. Moreover, the agents are often heterogeneous. The deployment of these new complex MAS requires new multiagent architectures and new solutions for the related problems of these systems such heterogeneity, openness and reliability.

To promote the success of the emerging agent-based applications, FIPA provides an abstract environment for the agent deployment and agent communication (see <http://www.fipa.org>). This environment implements a set of agents which provide the basic services for the deployment of MAS (see Figure 21.2).

The FIPA architecture offers several facilities to deploy MAS and to add dynamically new agents. However, several problems (fault, observation, etc.) have not been solved. Another way to deploy MAS, to achieve fault-tolerance and to solve other problems related to this deployment is to reuse solutions provided for distributed systems. For instance, replication of data and/or computation is an effective way to achieve fault tolerance in distributed systems. A replicated software component is defined as a software component that possesses a representation on two or more hosts (Guerraoui and Schiper, 1997). But in most cases, replication is decided by the programmer and applied stat-

ically, before the application starts. This works fine because the criticality of components (e.g., main servers) may be well identified and remain stable during the application session. Opposite to that, in the case of multiagent applications, the criticality of agents may evolve dynamically during the course of computation. Moreover, the available resources are often limited. Thus, simultaneous replication of all the agents of a large-scale system is not feasible. An idea is thus to automatically and dynamically apply replication mechanisms where (to which agents) and when it is most needed (Guessoum et al., 2002).

The provided solutions for MAS deployment are promising. The emergent applications (see section 4) allow to validate these solutions, answer the open questions and complete the existing methodologies to deal with the deployment.

5. Application Opportunities

MAS rely on several sub-fields of computer science such as object-oriented programming, artificial intelligence, artificial life, distributed and concurrent systems. The first applications of MAS have been used to improve existing systems in these sub-fields and to deal with their limitations. For example, MAS appear as interesting new tools to control complex process where information flow is abundant and alarms are common. These systems were often based on artificial intelligence techniques. A large wide of applications have therefore been developed in this area: air-traffic management to increase the efficiency or air travel, intensive care monitoring to assist the clinical staff in decision making. Moreover, the key concepts of MAS (emergence, self-organization, etc.) are very useful to understand/explain complex systems. A wide range of applications in multiagent simulation have thus been developed including bioinformatics, ecosystems and economic models.

Several recent emergent application domains are based on a set of distributed and cooperative entities which manage a large set of heterogeneous resources and provide services to the users. The management of this open set of resources is a hard problem, new resources can be dynamically added and existing ones can be changed and removed. Moreover, the interaction of various users to facilitate the use of this set of heterogeneous entities is not easy. For instance Grid Computing (see chapter 20), Ambient Intelligence and Web Services are the well known and promising emergent applications:

- Grid Computing intends to realize an infrastructure for large-scale distributed scientific applications.
- Ambient Intelligence embraces the advent of new computing systems, consisting of smart computing systems devices, which are likely to be more and more surrounded with in our working place, at home and during our leisure (Servat and Drogoul, 2002).

- Web services is a new way that adapts businesses to Internet technologies. The development of industry standards, products, and tools for supporting Web service system development is a very active area.

The approach to building Grid Computing platforms, Ambient Intelligence and Web services systems has several similarities with the engineering process of a collection of agents. For these applications, agents will be used to facilitate the design of applications. For examples, agents are used to

- Interact with users (personal assistant);
- Find and select components/services that match a given requirement; and
- Configure or compose the selected components/services.

Moreover, agents are used to control the execution of the so built systems and allow the self-configuration of the components/services to deal with dynamic changes.

6. A Roadmap for Agent-Oriented Software Engineering

Based on the examination of current status of agent technology, AgentLink published a roadmap of agent research and development over the first decade of the 21st century (Luck et al., 2003), consisting of four major phases. Currently we are in the first phase, where agent systems are usually built from scratch, with ad-hoc designs and little of reuse. Usually, the agent system is developed by just one team, for a particular application, in a concrete domain where the ontology and the communication protocols among agents are well-defined in advance. Agents are implemented with an object-oriented language (e.g., Java), sometimes with additions (such as a rule engine, a prolog interpreter or some other declarative language). There is not too much use of agent-oriented methodologies, and only in lucky situations object-oriented methods and tools are used. The resulting systems have some features of agency, such as a goal-driven architecture, the introduction of learning mechanisms (either individual or collective), some emergent behavior, or the definition of higher-level interactions (i.e., based on some agent communication language, such as FIPA ACL). The benefits of this first generation of agent systems is that some of them can show the potential of agent technology. But in order to transfer this to the industry there is a clear need of adopting well-established languages (for modeling and implementing agents), and a set of methods and tools to work with. The experience in the development of these agent-based systems should provide the foundations for well-established methodologies.

As it has been presented before, there are already several attempts to define these methodologies, see, e.g., (Giorgini et al., 2003; Giunchiglia et al.,

2003; Wooldridge et al., 2002c). Although most of them started from theoretical work (e.g., Gaia, Tropos), some are based on practical experience from real developments (e.g., MaSE, INGENIAS, PASSI). They are both evolving, the former by being finally used in practice, the latter by being formalized as experience provides more insight. There is also a trend to unify agent modeling languages, as it is the case of AUML, and for integrating different methods, for instance, by using meta-models, e.g., MetaMeth (Cossentino et al., 2003).

AgentLink expects that these methodologies will establish in the period 2003-05. This will have an impact in the adoption of agent technology, as agent systems, because of a more formal use of engineering practices, will gain in quality, scalability, robustness, reuse, and integration with legacy systems. The establishment of AOSE practices will facilitate coordination of agent developments by different teams, and will go together with the availability of agent platforms with more support for agent management, system scalability and robustness. Rather than creating new agent implementation languages, agent platforms will provide configurable frameworks for defining new types of agents, which will be instantiated by describing the main elements of their behavior with agent-related concepts, such as goal, task, rule, policy, etc. At this moment, it will be easier to quickly create and deploy new types of agents that will be able to interoperate, following a service-oriented architecture, with other agents in the system. This phase will therefore allow the development of agent systems, but some issues will be still pending to exploit full agent capabilities.

The third phase, during 2006-08 will promote a deeper integration and standardization of agent modeling language and development methodologies, as a result of the experience with the methodologies and their supporting tools. This will go together with advances in specific questions that the agent community is currently addressing, concerning the openness of agent systems, more specifically to deal with the semantic heterogeneity. At this time, the agent-oriented approach will be ready to have a higher relevance at industrial level. There will be a market of agent components, generic or domain-specific, which will be based on the use of open protocols and agent communication languages. These will be used as building blocks that the agents will be able to acquire dynamically depending on the situation. This will facilitate inter-domain interactions and higher degrees of service composition.

In the final phase, agents will further develop their learning capabilities, therefore it will be possible to develop complex coordination schemas and role assignment strategies. Organizations will be able to change dynamically and will be an important building block for the development of new MAS. There will be not only agents for sale, but complete, highly configurable, organizations of agents. In this phase, systems will be conceived as a set of interacting organizations.

7. Conclusions

AOSE has got the attention of many practitioners and researchers in the last five years. As experience in the development of agent-based software becomes more usual, more systematic approaches for building this kind of systems start to appear. They start usually from well-proved object-oriented methodologies, which are extended to cope with new concepts from agent technology, thus integrating techniques from other disciplines, specially from the field of artificial intelligence. Some lessons can be learned from those areas. For instance, the need for unification of terms and modelling language as a basis for the integration of methods and supporting tools. There are efforts in this direction, as it has been described in this chapter.

Agent technologies should not be considered as a totally revolutionary approach, but rather as an integration and extension of current state of the art. For instance, agents go further than component technology, increasing the levels of reusability to more complex entities, such as agents and organizations. They also provide new ways of distribution and flexibility of information processing, together with an inherent adaptation to changing environment. As such, the agent paradigm fits well with the needs for the coming wireless multi-modal information services. The current evolution of proposals in the area has shown the feasibility of the technology. Now it is time for industrial deployment and support, and in this sense the role of standards plays is relevant. A positive sign is that FIPA has already started activities in this line.

References

- [Abelson, 1996] Abelson, S. (1996). *Structure and Interpretation of Computer Programs*. The MIT Press.
- [Abelson et al., 2000] Abelson, H., Allen, D., Coore, D., Hanson, C., Homsy, G., Knight, T., Napal, R., Rauch, E., Sussmann, G., and Weiss, R. (2000). Amorphous Computing. *Communications of the ACM*, 43(5), pages 43–50.
- [Abowd, 1999] Abowd, G. D. (1999). Software Engineering Issues for Ubiquitous Computing. *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*, Los Angeles, USA.
- [Adorni et al., 2001] Adorni, G., Bergenti, F., Poggi, A., and Rimassa, G. (2001). Enabling FIPA Agents on Small Devices. *Proceedings of the 5th International Workshop on Cooperative Information Agents (CIA 2001)*, Rome, Italy.
- [Aksit et al., 1993] Aksit, M., Wakita, K., Bosch, J., Bergmans, L., and Yonezawa, A. (1993). Abstracting Object-Interactions using Composition-Filters. Guerraoui, R., Nierstrasz, O., Riveill, M. (Eds.) *Object-based Distributed Processing*, Springer-Verlag.
- [Alberts et al., 2001] Alberts, D., Garstka, J., Hayes, R., and Signori, D. (2001) Understanding Information Age Warfare. *CCRP Publication Series*.
- [Amant, 2003] Amant, R. S. (2003). Planning Resources at the North Carolina State University. Available at <http://www.csc.ncsu.edu/faculty/stamant/planning-resources.html>.
- [Ankolekar et al., 2002] Ankolekar, A., Burstein, M., Hobbs, J., Lassila, O., McDermott, D., Martin, D., McIlraith, S., Narayanan, S., Paolucci, M., Payne, T., and Sycara, K. (2002). DAML-S: Web Service Description for the Semantic Web. *Proceedings of the 1st International Semantic Web Conference (ISWC)*, pages 348–363.

- [Aridor and Lange, 1998] Aridor, Y., and Lange, D.B. (1998). Agent Design Patterns: Elements of Agent Application Design. *Proceedings of the 2nd International Conference on Autonomous Agents (Agents'98)*, pages 108–115, St. Paul, USA.
- [Ark and Selker, 1999] Ark, W. S., and Selker, T. (1999). A Look at Human Interaction with Pervasive Computers. *IBM Systems Journal*, 38(4).
- [Arnold et al., 1999] Arnold, K., O'Sullivan, B., Scheifler, R., Waldo, J., and Wollrath, A. (1999). *The Jini Specification*. Sun Microsystems.
- [Arquilla and Ronfeldt, 2000] Arquilla, J., and Ronfeldt, D. (2000). Swarming and the Future of Conflict. Available at <http://www.rand.org/publications/DB/DB311>.
- [Avila-Rosas et al., 2002] Avila-Rosas, A., Moreau, L., Dialani, V., Miles, S., and Liu, X. (2002). Agents for the Grid: A Comparison with Web Services (Part II: Service Discovery). *Proceedings of the Workshop on Challenges in Open Agent Systems*, Bologna, Italy.
- [Avouris and Gasser, 1992] Avouris, N. A., and Gasser, L. (Eds.) (1992). *Distributed Artificial Intelligence: Theory and Praxis*. Kluwer Academic Publishers.
- [Axelrod, 1984] Axelrod, R. (1984). *The Evolution of Cooperation*. Basic Books.
- [Baecker et al., 1997] Baecker, R., DiGiano, C., and Marcus, A. (1997). Software Visualization for Debugging. *Communications of the ACM*, 40(4), pages 44–54.
- [Ball, 1996] Ball, P. (1996). *The Self-Made Tapestry: Pattern Formation in Nature*. Princeton University Press.
- [Bardram, 1998] Bardram, J. (1998). Designing for the Dynamics of Cooperative Work Activities. *Proceedings of the ACA4 Conference on Computer Supported Cooperative Work*.
- [Barthelmess and Anderson, 2002] Barthelmess, P., and Anderson, K. M. (2002). A View of Software Development Environments based on Activity Theory. *Computer Supported Cooperative Work (CSCW)*, 11(1–2), pages 13–37.
- [Bauer, 1972] Bauer, F. L. (1972). Software Engineering. *Information Processing 71*.

- [Bauer, 1999] Bauer, B. (1999). Extending UML for the Specification of Interaction Protocols. Submission for the 6th Call for Proposal of FIPA and revised version of FIPA-99.
- [Bauer, 2001] Bauer, B. (2001). UML Class Diagrams Revisited in the Context of Agent-Based Systems. Wooldridge, M., Ciancarini, P., and Weiss, G. (Eds.) *Proceedings of Agent-Oriented Software Engineering (AOSE'01)*, pages 1–8.
- [Bauer et al., 2001] Bauer, B., Müller, 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), pages 207–230.
- [Bellavista et al., 2001] Bellavista, P., Corradi, A., and Stefanelli, C. (2001). Mobile Agent Middleware for Mobile Computing. *IEEE Computer*, 34(3).
- [Bellifemine et al., 2001] Bellifemine, F., Poggi, A., and Rimassa, G. (2001). Developing Multi-agent Systems with a FIPA-Compliant Agent Framework. *Software Practice and Experience*, 31, pages 103–128.
- [Benni, 1988] Beni, G. (1988). The Concept of Cellular Robotic System. *Proceedings of the IEEE International Symposium on Intelligent Control*, Los Alamitos, CA, pages 57–62.
- [Benni and Hackwood, 1992] Beni, G., and Hackwood, S. (1992). Stationary Waves in Cyclic Swarms. *Proceedings of the IEEE International Symposium on Intelligent Control*, Los Alamitos, CA, pages 234–242.
- [Benni and Wang, 1989] Beni, G., and Wang, J. (1989). Swarm Intelligence. *Proceedings of 7th Annual Meeting of the Robotics Society of Japan*, Tokyo, pages 425–428.
- [Benni and Wang, 1991] Beni, G., and Wang, J., 1991. Theoretical Problems for the Realization of Distributed Robotic Systems. *Proceedings of the IEEE International Conference on Robotic and Automation*, Los Alamitos, CA, pages 1914–1920.
- [Bergenti, 2003] Bergenti, F. (2003). A Discussion of Two Major Benefits of Using Agents in Software Development. *Engineering Societies in the Agents World III*, pages 1–12, Springer-Verlag.
- [Bergenti and Poggi, 2001] Bergenti, F., and Poggi, A. (2001). A Development Toolkit to Realize Autonomous and Interoperable Agents. *Proceedings of the 5th International Conference on Autonomous Agents*, pages 632–639.

- [Bergenti and Ricci, 2002] Bergenti, F., and Ricci, A. (2002). Three Approaches to the Coordination of Multiagent Systems. *Proceedings of the 2002 ACM Symposium on Applied Computing*, pages 367–372.
- [Berlin, 1994] Berlin A. (1994) *Towards Intelligent Structures: Active Control of Buckling*. Ph.D. Thesis, MIT.
- [Berners-Lee et al., 2001] Berners-Lee, T., Hendler, J., and Lassila, O. (2001). The Semantic Web. *Scientific American*, 284(5), pages 34–43.
- [Bernon et al., 2003] Bernon, C., Camps, V., Gleizes, M.-P., and Picard, G. (2003). Tools for Self-Organizing Applications Engineering. Di Marzo Serugendo, G., Karageorgos, A., Rana, O. F., and Zambonelli, F. (Eds.) *Proceedings of the 1st International Workshop on Engineering Self-Organizing Applications (ESOA 2003)*, Melbourne, Australia.
- [Bernon et al., 2002] Bernon, C., Gleizes, M.-P., Peyruqueou, S., and Picard, G. (2002). ADELFE: A Methodology for Adaptive Multi-Agent Systems Engineering. Petta, P., Tolksdorf, R., and Zambonelli, F. (Eds.) *Proceedings of the 3rd International Workshop on Engineering Societies in the Agents World (ESAW 2002)*, pages 156–169.
- [Biebricher, C. K., 1995] Biebricher, C. K., Nicolis, G., and Schuster, P. (1995). Self-Organization in the Physico-Chemical and Life Sciences. European Union, 1995.
- [Bieszczad et al., 1998] Bieszczad, A., Pagurek, B., and White, T. (1998). Mobile Agents for Network Management. *IEEE Communications Surveys*, 1(1), pages 2–9.
- [Bigus et al., 2002] Bigus, J. P., Schlonagle, D. A., Pilgrim, J. R., Mills III, W. N., and Diao, Y. (2002). ABLE: A Toolkit for Building Multiagent Autonomic Systems. *IBM Systems Journal*, 41(2), pages 350–371.
- [Binder and Lichtl, 2002] Binder, W., and Lichtl, B. (2002). Using a Secure Mobile Object Kernel as Operating System on Embedded Devices to Support the Dynamic Upload of Applications. *Proceedings of the 6th IEEE International Conference on Mobile Agents (MA'2002)*, Barcelona, Spain.
- [Bledsoe, 1985] Bledsoe, L. J., and Henschen W. W. (1985). What is Automated Theorem Proving? *Journal of Automated Reasoning*, 1(1), pages 23–28.
- [Boehm, 1984] Boehm, B. W. (1984). Verifying and Validating Software Requirements and Design Specifications. *IEEE Software*, 1(1), pages 75–84.

- [Boissier, 2003] Boissier, O. (2003). Master Web Intelligence: Organizations. Available at <http://www.emse.fr/~boissier>.
- [Bonabeau, 2003] Bonabeau, E. (2003). Swarm Intelligence. *Proceedings of Swarming: Network Enabled C4ISR*, Tysons Corner, VA.
- [Bonabeau et al., 1999] Bonabeau, E., Dorigo, M., and Theraulaz, G. (1999). *Swarm Intelligence*. Oxford University Press.
- [Booch, 1994] Booch, G. (1994). *Object-Oriented Analysis and Design with Applications*. Addison-Wesley.
- [Booch et al., 1999] Booch, G., Rumbaugh, J., and Jacobson, I. (1999). *The Unified Modeling Language User Guide*. Addison-Wesley.
- [Bordini et al., 2002] Bordini, R. H., Bazzan, A. L. C., de O. Jannone, R., Basso, D. M., Vicari, R. M., and Lesser, V. R. (2002). Agentspeak(XL): Efficient Intention Selection in BDI Agents via Decision-Theoretic Task Scheduling. *Proceedings of the 1st International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 1294–1302.
- [Borriello, 2002] Borriello, G. (2002). Key Challenges in Communication for Ubiquitous Computing. *IEEE Communications Magazine*.
- [Bowen, 2003] Bowen, Jonathan (2003). Formal Methods Resources. Available at <http://www.afm.sbu.ac.uk>.
- [Box et al., 2000] Box, D., Skonnard, A., and Lam, J. (2000). *Essential XML: Beyond Markup*. DevelopMentor Series. Addison-Wesley.
- [Bratman, 1987] Bratman, M. E. (1987). *Intentions, Plans, and Practical Reason*. Harvard University Press.
- [Bratman et al., 1988] Bratman, M. E., Israel, D., and Pollack, M. (1988). Plans and Resource-Bounded Practical Reasoning. *Journal of Computational Intelligence*, 4(4), pages 349–355.
- [Brazier et al., 1994] Brazier, F. M. T., van Langen, P., Treur, J., Wijngaards, N., and Willems, M. (1994). Modelling a Design Task in DESIRE: The VT Example. Technical Report IR-377, Universiteit Amsterdam.
- [Brazier et al., 1997] Brazier, F. M. T., Dunin-Keplicz, B. M., Jennings, N. R., and Treur, J. (1997). DESIRE: Modelling Multi-Agent Systems in a Compositional Formal Framework. *International Journal of Cooperative Information Systems*, 6(1), pages 67–94.

- [Brazier et al., 1999] Brazier, F. M. T., Jonker, C. M., Jungen, F. J., and Treur, J. (1999). Distributed Scheduling to Support a Call Centre: A Co-operative Multi-Agent Approach. *Applied Artificial Intelligence Journal*, 13.
- [Brazier et al., 2000] Brazier, F. M. T., Jonker, C. M., and Treur, J. (2000). Compositional Design and Reuse of a Generic Agent Model. *International Journal of Cooperative Information Systems*, 9(3), pages 171–207.
- [Brazier et al., 2002] Brazier, F. M. T., Jonker, C. M., and Treur, J. (2002). Principles of Component-Based Design of Intelligent Agents. *Data and Knowledge Engineering*, 41.
- [Bresciani et al., 2001] Bresciani, P., Perini, A., Giorgini, P., Giunchiglia, F., and Mylopoulos, J. (2001). A Knowledge Level Software Engineering Methodology for Agent Oriented Programming. *Proceedings of the 5th International Conference on Autonomous Agents*, pages 648–655, Montreal, CA.
- [Bresciani et al., 2002] Bresciani, P., Giorgini, P., Giunchiglia, F., Mylopoulos, J., and Perini, A. (2002). Tropos: An Agent-Oriented Software Development Methodology. Technical Report DIT-02-0015, University of Trento.
- [Brézillon, 2003] Brézillon, P. (2003). Focusing on Context in Human-Centered Computing. *IEEE Intelligent Systems*, 18(3).
- [Brueckner, 2000] Brueckner, S. A. (2000). Return from the Ant: Synthetic Ecosystems for Manufacturing Control. Dr.rer.nat. Thesis. Humboldt University.
- [Brueckner and Parunak, 2002] Brueckner, S. A., and Parunak, H. V. D. (2002). Swarming Agents for Distributed Pattern Detection and Classification. *Proceedings of Workshop on Ubiquitous Computing, AAMAS 2002*, Bologna, Italy.
- [Brueckner and Parunak, 2003] Brueckner, S. A., and Parunak, H. V. D. (2003). Information-Driven Phase Changes in Multi-Agent Coordination. *Proceedings of Autonomous Agents and Multi-Agent Systems (AAMAS 2003)*, pages 950–951.
- [Bryson, 2001] Bryson, J. J. (2001). *Intelligence by Design: Principles of Modularity and Coordination for Engineering Complex Adaptive Agents*. Ph.D. Thesis, MIT.
- [Bryson et al., 2002] Bryson, K., Luck, M., Joy, M., and Jones, D. (2002). Agent Interaction for Bioinformatics Data Management. *Applied Artificial Intelligence*.

- [Burmeister, 1996] Burmeister, B. (1996). Models and Methodology for Agent-Oriented Analysis and Design. *Working Notes of the KI'96 Workshop on Agent Oriented Programming and Distributed Systems*.
- [Burrafato and Cossentino, 2002] Burrafato, P., and Cossentino, M. (2002). Designing a Multi-Agent Solution for a Bookstore with the PASSI Methodology. *Proceedings of the 4th International Bi-Conference Workshop on Agent-Oriented Information Systems (AOIS-2002)*, Toronto, CA.
- [Busetta et al., 1998] Busetta, P., Rönquist, R., Hodgson, A., and Lucas, A. (1998). JACK Intelligent Agents – Components for Intelligent Agents in Java. Technical Report, Agent Oriented Software Pty. Ltd. Available at <http://www.agent-software.com>.
- [Busetta et al., 2001] Busetta, P., Carman, M., Serafini, L., Stockinger, K., and Zini, F. (2001). Grid Query Optimisation in the Data Grid. Technical Report IRST 0109-01, Istituto Trentino di Cultura.
- [Bush et al., 2001] Bush, G., Cranefield, S., and Purvis, M. (2001). The Styx Agent Methodology. The Information Science Discussion Paper Series 2001/02, University of Otago, New Zealand.
- [Busi et al., 2001] Busi, N., Ciancarini, P., Gorrieri, R., and Zavattaro, G. (2001). Coordination Models: A Guided Tour. Omicini, A., Zambonelli, F., Klusch, M., and Tolksdorf, R. (Eds.) *Coordination of Internet Agents: Models, Technologies, and Applications*, pages 6–24.
- [Bussmann, 1998] Bussmann, S. (1998). Agent-Oriented Programming of Manufacturing Control Tasks. *Proceedings of the 3rd International Conference on Multi-Agent Systems*, pages 57–63.
- [Butler et al., 2001] Butler, Z., Byrnes, S., and Rus, D. (2001). Distributed Motion Planning for Modular Robots with Unit-Compressible Modules. *Proceedings of the International Conference on Intelligent Robots and Systems*.
- [Butler et al., 2002] Butler, Z., Kotay, K., Rus, D., and Tomita, K. (2002). Generic Decentralized Control for a Class of Self-Reconfigurable Robots. *Proceedings of the IEEE International Conference on Robotics and Automation*.
- [Buyya et al., 2001] Buyya, R., Giddy, J., and Abramson, D. (2001). An Economy Grid Architecture for Service-Oriented Grid Computing. *Proceedings of the 10th IEEE International Heterogeneous Computing Workshop (HCW 2001)*, San Francisco, USA.

- [Caire et al., 2001a] Caire, G., Chainho, P., Evans, R., Garijo, F., Gomez Sanz, J., Kearney, P., Leal, F., Massonet, P., Pavon, J., and Stark, J. (2001a). Methodology for Agent Oriented Software Engineering. EURESCOM Project P907 Deliverable 3.
- [Caire et al., 2001b] Caire, G., Leal, F., Chainho, P., Evans, R., Garijo, F., Gomez-Sanz, J. J., Pavon, J., Kerney, P., Stark, J., and Massonet, P. (2001b). Agent Oriented Analysis using MESSAGE/UML. Weiss, G., Cianciarini, P., and Wooldridge, M. (Eds.) *Agent-Oriented Software Engineering II*, Springer-Verlag.
- [Camazine et al., 2001] Camazine, S., Deneubourg, J.-L., Franks, N. R., Sneyd, J., Theraulaz, G., and Bonabeau, E. (2001). *Self-Organization in Biological Systems*. Princeton University Press.
- [Camps et al., 1998] Camps, V., Gleizes, M.-P., and Glize, P. (1998). Une Théorie des Phénomènes Globaux Fondée sur des Interactions Locales. *Actes des Sixième Journées Francophones IAD&SMA (JFIADSMA98)*, Editions Hermès.
- [Capera et al., 2003a] Capera, D., Georgé, J.-P., Gleizes, M.-P., and Glize, P. (2003a). The AMAS Theory for Complex Problem Solving based on Self-Organizing Cooperative Agents. *Proceedings of the 1st International Workshop on Theory and Practice of Open Computational Systems (TAPOCS)*, pages 383–388.
- [Capera et al., 2003b] Capera, D., Georgé, J.-P., Gleizes, M.-P., and Glize, P. (2003b). Emergence of organisations, emergence of functions. *Proceedings of the FAISB'03 Symposium on Adaptive Agents and Multi-Agent Systems*.
- [Carriero and Gelernter, 1989] Carriero, N., and Gelernter, D. (1989). Linda in Context. *Communications of the ACM*, 32(4), pages 444–458.
- [Castelfranchi, 1998] Castelfranchi, C. (1998). Modelling Social Action for AI Agents. *Artificial Intelligence*, 103(1).
- [Castelfranchi, 2000] Castelfranchi, C. (2000). Founding Agent's 'Autonomy' on Dependence Theory. *Proceedings of 14th European Conference on Artificial Intelligence*, pages 353–357, Berlin, Germany.
- [Castelfranchi and Falcone, 1998] Castelfranchi, C., and Falcone, R. (1998). Towards a Theory of Delegation for Agent-Based Systems. *Robotics and Autonomous Systems*, 24:141.
- [Castro et al., 2001] Castro, J., Kolp, M., and Mylopoulos, J. (2001). A requirements-driven development methodology. Dittrich, K.R., Geppert, A.

- and Norrie, M.C., editors, *Proceedings of the 13th International Conference on Advanced Information Systems Engineering (CAiSE'01)*, volume 2068, pages 108–123, Interlaken, Switzerland. Springer-Verlag.
- [Castro et al., 2002] Castro, J., Kolp, M., and Mylopoulos, J. (2002). Towards Requirements-Driven Information Systems Engineering: The Tropos Project. *Information Systems*. Elsevier.
- [Cernuzzi and Rossi, 2002] Cernuzzi, L., and Rossi, G. (2002). On the Evaluation of Agent Oriented Modeling Methods. *Proceedings of the OOPSLA 2002 Workshop on Agent-Oriented Methodologies*, pages 21–30, Seattle, USA.
- [Chen and Finin, 2002] Chen, H., and Finin, T. (2002). Beyond Distributed AI – Agent Teamwork in Ubiquitous Computing. *Proceedings of the 1st International Workshop on Ubiquitous Agents on Embedded, Wearable, and Mobile Devices*, Bologna, Italy.
- [Cheung et al., 1997] Cheung, P., Berlin, A., Biegelsen, D. K., and Jackson, W. B. (1997). Batch Fabrication of Pneumatic Valve Arrays by Combining MEMS with Printed Circuit Board Technology. *Proceedings of the Symposium on Micro-Mechanical Systems*.
- [Chopra and Singh, 2003] Chopra, A., and Singh, M. P. (2003). Nonmonotonic Commitment Machines. Dignum, F. (Ed.) *Proceedings of the 2003 AAMAS Workshop on Agent Communication Languages*, Springer-Verlag.
- [Chung et al., 2000] Chung, L., Nixon, B. A., Yu, E., and Mylopoulos, J. (2000). *Non-Functional Requirements in Software Engineering*. Kluwer Academic Publishers.
- [Ciancarini et al., 2000] Ciancarini, P., Omicini, A., and Zambonelli, F. (2000). Multiagent System Engineering: The Coordination Viewpoint. Jennings, N. R., and Lespérance, Y. (Eds.) *Intelligent Agents VI. Agent Theories, Architectures, and Languages*, pages 250–259. Springer-Verlag.
- [Cimatti et al., 2002] Cimatti, A., Clarke, E. M., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., and Tacchella, A. (2002). NuSMV 2: An Opensource Tool for Symbolic Model Checking. *Computer Aided Verification*, Springer-Verlag.
- [Clarke et al., 1999] Clarke, E., Grumberg, O., and Peled, D. (1999). *Model Checking*. The MIT Press.
- [Clarke et al., 2001] Clarke, D., Elien, J.-E., Ellison, C., Fredette, M., Morcos, A., and Rivest, R. L. (2001). Certificate Chain Discovery in SPKI/SDSI. *Journal of Computer Security*, 9(4), pages 285–338.

- [Clearwater, 1996] Clearwater, S. H. (Ed.) (1996). *Market-Based Control. A Paradigm for Distributed Resource Allocation*. World Scientific Publishing.
- [Clements, 1996] Clements, P. C. (1996). A Survey of Architecture Description Languages. *Proceedings of the 8th International Workshop on Software Specification and Design*.
- [Clough, 2003] Clough, B. (2003). Emergent Behavior (Swarming): Tool Kit for Building UAV Autonomy. *Proceedings of Swarming: Network Enabled C4ISR*, Tysons Corner, VA.
- [Coffman et al., 1971] Coffman, E. G., Elphick, M., and Shoshani, A. (1971). System Deadlocks. *ACM Computing Surveys*, 3(2), pages 67–78.
- [Cohen, 1988] Cohen, J. (1988). A View of the Origins and Development of Prolog. *Communications of the ACM*, 31(1), pages 26–36.
- [Cohen and Levesque, 1990] Cohen, P. R., and Levesque, H. J. (1990). Intention is Choice with Commitment. *Artificial Intelligence*, 42, pages 213–261.
- [Cohen and Levesque, 1991] Cohen, P. R., and Levesque, H. J. (1991). Teamwork. *Nous*, 25(4), pages 487–512.
- [Coleman et al., 1994] Coleman, D., Arnold, P., Bodoff, S., Dollin, D., Gilchrist, H., Hayes, F., and Jeremas, P. (1994). *Object-Oriented Development: The FUSION Method*. Prentice-Hall International.
- [Collinot et al., 1996] Collinot, A., Drogoul, A., and Benhamou, P. (1996). Agent Oriented Design of a Soccer Robot Team. *Proceedings of ICMAS'96*.
- [Collis and Ndumu, 1999] Collis, J. C., and Ndumu, D. T. (1999). *The Role Modelling Guide*. Applied Research and Technology, BT Labs.
- [Conte et al., 1998] Conte, R., Gilbert, N., and Simao Sichman, J. (1998). MAS and Social Simulation: A Suitable Commitment. Sichman, J. S., Conte, R., and Gilbert, N. (Eds.) *Proceedings of the 1st International Workshop on Multi Agent Based Simulation*, pages 1–9. Springer-Verlag.
- [Conte and Sichman, 1995] Conte, R., and Sichman, J. S. (1995). Depnet: How to Benefit from Social Dependence. *Journal of Mathematical Sociology*, 20(2-3), pages 161–177.
- [Coore, 1999] Coore, D. (1999). Botanical Computing: A Developmental Approach to Generating Interconnect Topologies on an Amorphous Computer. Ph.D. Thesis, MIT.

- [Cossentino, 2001] Cossentino, M. (2001). Different Perspectives in Designing Multi-Agent System. *Proceedings of the Workshop on Agent Technology and Software Engineering*, Erfurt, Germany.
- [Cossentino and Potts, 2002] Cossentino, M., and Potts, C. (2002). A CASE Tool Supported Methodology for the Design of Multi-Agent Systems. *Proceedings of the 2002 International Conference on Software Engineering Research and Practice*, Las Vegas, USA.
- [Cossentino et al., 2003] Cossentino, M., Hopmans, G., and Odell, J. (2003). FIPA standardization Activities in the Software Engineering Area. *Proceedings of the 2003 Workshop on Objects and Agents (WOA03)*, Cagliari, Italy.
- [Cox et al., 2001] Cox, S. J., Fairman, M. J., Xue, G., Wason, J. L., and Keane, A. J. (2001). The Grid: Computational and Data Resource Sharing in Engineering Optimisation and Design Search. *Proceedings of the 2001 ICPP Workshops*, pages 207–212.
- [Crutchfield, 1994] Crutchfield, J. P. (1994). The Calculi of Emergence: Computation, Dynamics, and Induction. *Physica D*, 75, pages 11–54.
- [Curbera et al., 2002a] Curbera, F., Duftler, M., Khalaf, R., Nagy, W., Mukhi, N., and Weerawarana, S. (2002a). Unraveling the Web Services Web. *IEEE Internet Computing*, 6, pages 86–93.
- [Curbera et al., 2002b] Curbera, F., Goland, Y., Klein, J., Leymann, F., Roller, D., Thatte, S., and Weerawarana, S. (2002b). Business Process Execution Language for Web Services. Available at <http://www.ibm.com/developerworks/library/ws-bpel>.
- [Dam, 2003] Dam, K. H. (2003). Evaluating Agent-Oriented Software Engineering Methodologies. Master's Thesis, RMIT University.
- [Dam and Winikoff, 2003] Dam, K. H., and Winikoff, M. (2003). Comparing Agent-Oriented Methodologies. *Proceedings of the 5th International Bi-Conference Workshop on Agent-Oriented Information Systems*.
- [Dardenne et al., 1993] Dardenne, A., van Lamsweerde, A., and Fickas, S. (1993). Goal-Directed Requirements Acquisition. *Science of Computer Programming*, 20(1–2), pages 3–50.
- [Darimont et al., 1997] Darimont, R., Delor, E., Massonet, P., and van Lamsweerde, A. (1997). GRAIL/KAOS: An Environment for Goal-Driven Requirements Engineering. *Proceedings of the 19th International Conference on Software Engineering*, pages 612–613.

- [Dastani et al., 2003] Dastani, M., van Riemsdijk, B., Dignum, F., and Meyer, J. J. (2003). A Programming Language for Cognitive Agents: Goal Directed 3APL. *Proceedings of the 1st Workshop on Programming Multiagent Systems: Languages, Frameworks, Techniques, and Tools (ProMAS03)*.
- [Davis and Smith, 1983] Davis, R., and Smith, R. G. (1983). Negotiation as a Metaphor for Distributed Problem-Solving. *Artificial Intelligence*, 20, pages 63–109.
- [Day and Lawrence, 2000] Day, S. J., and Lawrence, P. A. (2000). Measuring Dimensions: The Regulation of Size and Shape. *Development*, 127, 2977–2987.
- [Dayal et al., 2001] Umeshwar, D., Meichun, H., and Ladin, R. (2001). Business Process Coordination: State of the Art, Trends and Open Issues. Apers, P. M. G., Atzeni, P., Ceri, S., Paraboschi, S., Ramamohanarao, K., and Snodgrass, R. T. (Eds.) *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB 2001)*, pages 3–13.
- [De Giacomo et al., 2000] De Giacomo, G., Lesperance, Y., and Levesque, H. J. (2000). Congolog, A Concrrent Programming Language based on the Situation Calculus. *Artificial Intelligence*, 121, pages 109–169.
- [De Roure et al., 2003] De Roure, D., Jennings, N. R., and Shadbolt, N. (2003). The Semantic Grid: A Future e-Science Infrastructure. Berman, F., Fox, G., and Hey, A. J. G. (Eds.) *Grid Computing: Making the Global Infrastructure a Reality*, pages 437–470, John Wiley & Sons.
- [Debenham and Henderson-Sellers, 2002] Debenham, J., and Henderson-Sellers, B. (2002). Full Lifecycle Methodologies for Agent-Oriented Systems – The Extended OPEN Process Framework. *Proceedings of Agent-Oriented Information Systems (AOIS-2002)*, Toronto, CA.
- [Decker, 1995] Decker, K. S. (1995). *Environment Centered Analysis and Design of Coordination Mechanisms*. Ph.D. Thesis, University of Massachusetts.
- [Decker, 1996a] Decker, K. S. (1996a). TÆMS: A Framework for Environment Centered Analysis and Design of Coordination Mechanisms. O’Hare, G. M. P., and Jennings, N. R., (Eds.) *Foundations of Distributed Artificial Intelligence*, pages 429–448, John Wiley & Sons.
- [Decker, 1996b] Decker, K. S. (1996b). Task Environment Centered Simulation. *Simulating Organizations: Computational Models of Institutions and Groups*. AAAI Press/The MIT Press.

- [Decker et al., 1989] Decker, K. S., Durfee, E. H., and Lesser, V. R. (1989). Evaluating Research in Cooperative Distributed Problem Solving. Huhns, M. N., and Gasser, L. (Eds.) *Distributed Artificial Intelligence, Volume 2*, pages 487–519. Pitman/Morgan Kaufmann.
- [Decker et al., 1997] Decker, K. S., Sycara, K., and Williamson, M. (1997). Middle-Agents for the Internet. *Proceedings of IJCAI'97*.
- [Decker et al., 2001] Decker, K. S., Zheng, X., and Schmidt, C. (2001). A Multi-Agent System for Automated Genetic Annotation. *Proceedings of the 5th ACM International Conference on Autonomous Agents*, Montreal, Canada.
- [DeLoach, 2001] DeLoach, S. A. (2001). Analysis and Design using MaSE and agenTool. *Proceedings of the 12th Midwest Artificial Intelligence and Cognitive Science Conference (MAICS)*, Miami University Press.
- [DeLoach, 2002] DeLoach, S. A. (2002). Modeling Organizational Rules in the Multiagent Systems Engineering Methodology. *Proceedings of the 15th Canadian Conference on Artificial Intelligence*.
- [DeLoach and Wood, 2001] DeLoach, S. A., and Wood, M. (2001). Developing Multiagent Systems with agentTool. Castelfranchi, C., and Lesperance, Y. (Eds.) *Intelligent Agents VII.*, pages 46–60, Springer-Verlag.
- [DeLoach et al., 2001] DeLoach, S. A., Wood, M. F., and Sparkman, C. H. (2001). Multiagent Systems Engineering. *International Journal of Software Engineering and Knowledge Engineering*, 11(3), pages 231–258.
- [DeLoach et al., 2003] DeLoach, S. A., Matson, E. T., and Li, Y. (2003). Exploiting Agent Oriented Software Engineering in Cooperative Robotics Search and Rescue. *International Journal of Pattern Recognition and Artificial Intelligence*.
- [Demazeau, 1995] Demazeau, Y. (1995). From Cognitive Interactions to Collective Behaviour in Agent-Based Systems. *Proceedings of the European Conference on Cognitive Science*, pages 117–132, Saint-Malo, France.
- [Denti et al., 2002] Denti, E., Omicini, A., and Ricci, A. (2002). Coordination Tools for MAS Development and Deployment. *Applied Artificial Intelligence*, 16(9-10), pages 721–752.
- [Depke et al., 2001] Depke, R., Heckel, R., and Kuster, J. M. (2001). Improving the Agent-Oriented Modeling Process by Roles. *Proceedings of the 5th International Conference on Autonomous Agents*, pages 640–647.

- [Devedzic, 1999] Devedzic, V. (1999). A Survey of Modern Knowledge Modeling Techniques. *Expert Systems with Applications*, 17(275).
- [Devlin, 2003] Devlin, K. (2003). Why Universities Require Computer Science Students to take Math. *Communications of the ACM*, 46(9), pages 37–39.
- [Dey, 2001] Dey, A. K. (2001). Evaluation of Ubiquitous Computing Systems: Evaluating the Predictability of Systems. Abowd, G. D., Brumitt, B., and Shafer, S. (Eds.) *Proceedings of Evaluation Methods for Ubiquitous Computing Workshop*, Springer-Verlag.
- [Dietterich, 1998] Dietterich, T. G. (1998). Machine-Learning Research: Four Current Directions. *AI Magazine*, 18(4), pages 97–136.
- [DiLeo et al., 2002] DiLeo, J., Jacobs, T., and DeLoach, S. (2002). Integrating Ontologies into Multiagent Systems Engineering. *Proceedings of the 4th International Bi-Conference Workshop on Agent-Oriented Information Systems*.
- [Do et al., 2003] Do, T., Kolp, M., and Pirotte, A. (2003). Social Patterns for Designing Multi-agent Systems. *Proceedings of the 15th International Conference on Software Engineering and Knowledge Engineering (SEKE'03)*.
- [Dorigo and Di Caro, 1999] Dorigo, M., and Di Caro, G. (1999). The Ant Colony Optimization Meta-Heuristic. Corne, D., Dorigo, M., and Glover, F. (Eds.) *New Ideas in Optimization*, McGraw-Hill.
- [Dorigo et al., 1996] Dorigo, M., Maniezzo, V., and Colorni, A. (1996). The Ant System: Optimization by a Colony of Cooperating Agents. *IEEE Transactions on Systems, Man, and Cybernetics*, 26(1), pages 1–13.
- [Drogoul and Zucker, 1998] Drogoul, A., and Zucker, J. (1998). Methodological Issues for Designing Multi-Agent Systems with Machine Learning Techniques: Capitalizing Experiences from the Robocup Challenge. Technical Report LIP6 1998/041, Laboratoire d'Informatique de Paris 6.
- [Du Bois, 1997] Du Bois, P. (1997). The ALBERT II Reference Manual. Technical Report RR-97-002, University of Namur.
- [Dubois, 1998] Dubois, P., and Heymans E. (1998). Scenario-Based Techniques for Supporting the Elaboration and the Validation of Formal Requirements. Technical Report CREWS 98-15, Université de Namur.
- [Dubois et al., 1994] Dubois, E., Du Bois, P., Dubru, F., and Petit, M. (1994). Agent-Oriented Requirements Engineering: A Case Study using the ALBERT Language. *Proceedings of the 4th International Working Conference*

- on *Dynamic Modelling and Information Systems (DYNMOD'94)*, pages 205–238.
- [Dulay et al., 2001] Dulay, N., Damianou, N., Lupu, E., and Sloman, M. (2001). A Policy Language for the Management of Distributed Agents. Wooldridge, M. J., Weiss, G., and Ciancarini, P. (Eds.) *Agent-Oriented Software Engineering II*, pages 84–100, Springer-Verlag.
- [Durfee and Lesser, 1991] Durfee, E. H., and Lesser, V. R. (1991). Partial Global Planning: A Coordination Framework for Distributed Hypothesis Formation. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(5), pages 1167–1183.
- [Durfee et al., 1989] Durfee, E. H., Lesser, V. R., and Corkill, D. D. (1989). Trends in Cooperative Distributed Problem Solving. *IEEE Transactions on Knowledge and Data Engineering*, 1(1), pages 63–83.
- [Edmonds, 1998] Edmonds, B. (1998). Social Embeddedness and Agent Development. *Proceedings of UKMAS'98*.
- [Edmonds, 2003a] Edmonds, B. (2003a). Simulation and Complexity – How They can Relate. Feldmann, V., and Mühlfeld, K. (Eds.) *Virtual Worlds of Precision*, Lit Verlag.
- [Edmonds, 2003b] Edmonds, B. (2003b). Against: A Priori Theory, For: Descriptively Adequate Computational Modelling. *The Crisis in Economics: The Post-Autistic Economics Movement: The first 600 days*, pages 175–179, Routledge.
- [Edwards, 2000] Edwards, S. J. A. (2000). Swarming on the Battlefield: Past, Present, and Future. Technical Report MR-1100-OSD, RAND.
- [Edwards, 2003] Edwards, S. J. A. (2003). Military History of Swarming. *Proceedings of Swarming: Network Enabled C4ISR*, Tysons Corner, VA.
- [Edwards and Grinter, 2001] Edwards, W. K., and Grinter, R. E. (2001). At Home with Ubiquitous Computing: Seven Challenges. Abowd, G. D., Brumitt, B., and Shafer, S. (Eds.) *Proceedings of the 3rd International Conference on Ubiquitous Computing (UBICOMP'2001)*, Springer-Verlag.
- [Ekudden et al., 2001] Ekudden, E., Horn, H., Melander, M., and Olin, J. (2001). On-Demand Mobile Media – A Rich Service Experience for Mobile Users. *Ericsson Review, The Telecommunications Technology Journal*, 4.
- [Elammari and Lalonde, 1999] Elammari, M., and Lalonde, W. (1999). An Agent-Oriented Methodology: High-Level and Intermediate Models. Wagner, G., and Yu, E. (Eds.) *Proceedings of the 1st International Workshop on Agent-Oriented Information Systems*.

- [Engeström et al., 1997] Engeström, Y., Brown, K., Christopher, C. L., and Gregory, J. (1997). Coordination, Cooperation, and Communication in the Courts: Expansive Transitions in Legal Work. Cole, M., Engeström, Y., and Vasquez, O. (Eds.) *Mind, Culture, and Activity*, Cambridge University Press.
- [Esler et al., 1999] Esler, M., Hightower, J., Anderson, T., and Borriello, G. (1999). Next Century Challenges: Data-Centric Networking for Invisible Computing – The Portolano Project at the University of Washington. *Proceedings of the ACM International Conference on Mobile Computing (MobiCOM'1999)*, Seattle, USA.
- [Esteva et al., 2002] Esteva, M., de la Cruz, D., and Sierra, C. (2002). Islander: An Electronic Institutions Editor. *Proceedings of the 1st International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 1045–1052.
- [Fallah-Seghrouchni and Suna, 2003] Fallah-Seghrouchni, A., and Suna, A. (2003). A Programming Language for Autonomous and Mobile Agents. *Proceedings of IAT 2003*.
- [Fankhauser et al., 1991] Fankhauser, P., Kracker, M., and Neuhold, E. J. (1991). Semantic vs. Structural Resemblance of Classes. *ACM SIGMOD RECORD* 20(4), pages 59–63.
- [Fano and Gershman, 2002] Fano, A., and Gershman, A. (2002). The Future of Business Services in the Age of Ubiquitous Computing. *Communications of the ACM*, 45(12).
- [Fensel and Motta, 2001] Fensel, D., and Motta, E. (2001). Structured Development of Problem Solving Methods. *Knowledge and Data Engineering*, 13(6), pages 913–932.
- [Fenster et al., 1995] Fenster, M., Kraus, S., and Rosenschein, J. S. (1995). Coordination without Communication: Experimental Validation of Focal Point Techniques. *Proceedings of International Conference on Multi-Agent Systems (ICMAS'95)*, pages 102–108, San Francisco, USA.
- [Ferber, 1999] Ferber, J. (1999). *Multi-Agent Systems*. Addison-Wesley.
- [Ferber and Gutknecht, 1998] Ferber, J., and Gutknecht, O. (1998). A Meta-Model for the Analysis and Design of Organizations in Multi-Agent Systems. *Proceedings of the 3rd International Conference on Multi-Agent Systems (ICMAS-98)*, pages 128–135.

- [Ferguson, 1992] Ferguson, I. A. (1992). *TouringMachines: An Architecture for Dynamic, Rational, Mobile Agents*. Ph.D. Thesis, University of Cambridge.
- [Fikes and Nilsson, 1971] Fikes, R., and Nilsson, J. (1971). STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence*, 2(3–4).
- [Fisher, 1994] Fisher, M. (1994). A Survey of Concurrent METATEM – The Language and its Applications. Gabbay, D. M., and Ohlbach, H. J. (Eds.) *Proceedings of the 1st International Conference on Temporal Logic*, pages 480–505, Springer-Verlag.
- [Fisher, 1995] Fisher, M. (1995). Representing and Executing Agent-Based Systems. Wooldridge, M., and Jennings, N. R. (Eds.) *Intelligent Agents: Theories, Architectures, and Languages*, pages 307–323. Springer-Verlag.
- [Fisher and Wooldridge, 1997] Fisher, M., and Wooldridge, M. J. (1997) On the Formal Specification and Verification of Multi-Agent Systems. *International Journal of Cooperative Information Systems*, 6:1, pages 37–65.
- [Fornara and Colombetti, 2002] Fornara, N., and Colombetti, M. (2002). Operational Specification of a Commitment-Based Agent Communication Language. *Proceedings of the 1st International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 535–542.
- [Forrest, 1991] Forrest, S. (Ed.) (1991). Emergent Computation: Self-Organizing, Collective, and Cooperative Phenomena in Natural and Artificial Computing Networks. Special issue of *Physica D*, The MIT Press.
- [Foster, 2002] Foster, I. (2002). What is the Grid? A Three Point Checklist. Available at <http://www-fp.mcs.anl.gov/~foster>.
- [Foster and Kesselman, 1999] Foster, I., and Kesselman, C. (Eds.) (1999). *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufman.
- [Foster et al., 2001] Foster, I., Kesselman, C., and Tuecke, S. (2001). The Anatomy of the Grid. Enabling Scalable Virtual Organizations. *International Journal of Supercomputer Applications*.
- [Foster et al., 2002] Foster, I., Kesselman, C., Nick, J., and Tucke, S. (2002). The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. *Open Grid Service Infrastructure WG*, Global Grid Forum.
- [Fox and Gruninger, 1998] Fox, M. S., and Gruninger, M. (1998). Enterprise Modelling. *AI Magazine*, 19(3), pages 109–121.

- [Fredriksson and Gustavsson, 2003] Fredriksson, M., and Gustavsson, R. (2003). Articulation of an Open Computational System for Network-Centric Warfare. Bubenko Jr., J. (Ed.) *Conference for the Promotion of Research in Information Technology at New Universities and at University Colleges (ITL)*, Visby, Sweden.
- [Fredriksson et al., 2003] Fredriksson, M., Gustavsson, R., and Ricci, A. (2003). Sustainable Coordination. Klusch, M., Bergamaschi, S., Edwards, P., and Petta, P. (Eds.) *Intelligent Information Agents: An AgentLink Perspective*, pages 203–233. Springer-Verlag.
- [Friedman-Hill, 2003] Friedman-Hill, E. (2003). Java Expert System Shell (JESS). Available at <http://herzberg.ca.sandia.gov/jess>.
- [Furmento et al., 2001] Furmento, N., Newhouse, S., and Darlington, J. (2001). Building Computational Communities from Federated Resources. *Proceedings of the 7th International Euro-Par Conference (Euro-Par 2001)*, pages 855–863.
- [Fuxman, 2001] Fuxman, A. (2001). Formal Analysis of Early Requirements Specifications. Master's Thesis, University of Toronto.
- [Fuxman et al., 2001] Fuxman, A., Pistore, M., Mylopoulos, J., and Traverso, P. (2001). Model Checking Early Requirements Specifications in Tropos. *Proceedings of the IEEE International Symposium on Requirements Engineering*, pages 174–181, Toronto, CA.
- [Fuxman et al., 2003] Fuxman, A., Liu, L., Pistore, M., Roveri, M., and Mylopoulos, J. (2003). Specifying and Analyzing Early Requirements in Tropos: Some Experimental Results. *Proceedings of the 11th IEEE International Requirements Engineering Conference*, Monterey Bay, USA.
- [Gamma et al., 1995] Gamma, E., Helm, R., Johnson, J., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.
- [Garcia-Molina and Germano, 1984] Garcia-Molina, H., Germano, F. (1984). Debugging a Distributed Computer System. *IEEE Transactions on Software Engineering*, 10(2), pages 210–219.
- [Garijo et al., 1998] Garijo, F., Tous, J., Matias, J. M., Corley, S., and Tessa-laar, M. (1998). Development of a Multi-Agent System for Cooperative Work with Network Negotiation Capabilities. Albayrak, S. (Ed.) *Intelligent Agents for Telecommunication Applications*, pages 204–219, Springer-Verlag.

- [Garijo et al, 2001] Garijo, F., Gómez-Sanz, J., Pavón, J., Massonet, P. Multi-Agent System Organisation. An Engineering Perspective. Proceedings of MAAMAW 2001, Springer-Verlag.
- [Gasser, 2001] Gasser, L. (2001). MAS Infrastructure Definitions, Needs, Prospects. Wagner, T., and Rana, O. (Eds.) *Infrastructure for Agents, Multi-Agent Systems, and Scalable Multi-Agent Systems*, pages 1–11. Springer-Verlag.
- [Gasser and Briot, 1992] Gasser, L., and Briot, J.-P. (1992). Distributed Artificial Intelligence: Theory and Praxis. *Object-Oriented Concurrent Programming and Distributed Artificial Intelligence*, pages 81–108. Kluwer Academic Publishers.
- [Gelernter and Carriero, 1992] Gelernter, D., and Carriero, N. (1992). Coordination Languages and Their Significance, *Communications of the ACM*, 25(2), pages 97–107.
- [Genesereth and Ketchpel, 1997] Genesereth, M. R., and Ketchpel, S. P. (1997). Software Agents. *Communications of the ACM*, 37(7).
- [Genesereth and Nilsson, 1987] Genesereth, M. R., and Nilsson, N. J. (1987). *Logical Foundations of Artificial Intelligence*. Morgan Kaufmann Publisher.
- [Genesereth et al., 1986] Genesereth, M. R., Ginsburg, M., and Rosenschein, J. S. (1986). Cooperation without Communication. *Proceedings of the National Conference on Artificial Intelligence (AAAI'86)*, pages 51–57, AAAI Press.
- [Georgé et al., 2003] Georgé, J.-P., Gleizes, M.-P., Glize, P., and Régis, C. (2003). Real-Time Simulation for Flood Forecast: An Adaptive Multi-Agent System STAFF. Kazakov, D., Kudenko, D., and Alonso, E. (Eds.) *Proceedings of the AISB'03 Symposium on Adaptive Agents and Multi-Agent Systems (AAMAS'03)*, pages 109–114.
- [Gervais, 2003] Gervais, M.-P. (2003). ODAC: An Agent-Oriented Methodology based on ODP. *Journal of Autonomous Agents and Multi-Agent Systems*, 7(3), pages 199–228.
- [Gervais and Muscutariu, 2001] Gervais, M.-P., and Muscutariu, F. (2001). Towards an ADL for Designing Agent-Based Systems. Wooldridge, M. J., Weiss, G., and Cianciarini, P. (Eds.) *Agent-Oriented Software Engineering II*, pages 263–277. Springer-Verlag.
- [Giampapa and Sycara, 2002] Giampapa, J. A., and Sycara, K. (2002). Team-Oriented Agent Coordination in the RETSINA Multi-Agent System. Technical Report CMU-RI-TR-02-34, Carnegie Mellon University.

- [Gilber, 1995] Gilbert, G. N., 1995. Emergence in Social Simulation. Gilbert, G. N., and Conte, R. (Eds.) *Artificial Societies: The Computer Simulation of Social Life*, UCL Press.
- [Giorgini et al., 2002] Giorgini, P., Nicchiarelli, E., Mylopoulos, J., and Sebastiani, R. (2002). Reasoning with Goal Models. *Proceedings of the International Conference of Conceptual Modeling*, Springer-Verlag.
- [Giorgini et al., 2003] Giorgini, P., Mueller, J., and Odell, J. (Eds.) (2003). *Agent-Oriented Software Engineering III*, Springer-Verlag.
- [Giunchiglia et al., 2002] Giunchiglia, F., Mylopoulos, J., and Perini, A. (2002). The Tropos Software Development Methodology: Processes, Models and Diagrams. *Proceedings of Agent-Oriented Software Engineering*, Springer-Verlag.
- [Giunchiglia et al., 2003] Giunchiglia, F., Odell, J., and Weiss, G. (Eds.) (2003). *Agent-Oriented Software Engineering IV*, Springer-Verlag.
- [Glaser, 1996] Glaser, N. (1996). The CoMoMAS Methodology and Environment for Multi-Agent System Development. Zhang, C., and Lukose, D. (Eds.) *Multi-Agent Systems Methodologies and Applications*, pages 1–16. Springer-Verlag.
- [Gleizes et al., 1999] Gleizes, P.-P., Camps, V., and Glize, P. (1999). A Theory of Emergent Computation Based on Cooperative Self-Organization for Adaptive Artificial Systems. *Proceedings of the 4th European Congress of Systems Science*, Valencia, Spain.
- [Goldstein, 1999] Goldstein, J. (1999). Emergence as a Construct: History and Issues. *Emergence*, 1-1, pages 49–72.
- [Gomez-Sanz and Fuentes, 2002] Gomez-Sanz, J., and Fuentes, R. (2002). Agent Oriented System Engineering with INGENIAS. *Proceedings of the 4th Iberoamerican Workshop on Multi-Agent Systems*.
- [Gomez-Sanz and Pavon, 2003] Gomez-Sanz, J., and Pavon, J. (2003). Agent Oriented Software Engineering with INGENIAS. Marík, V., Müller, J., and Pechoucek, M. (Eds.) *Multi-Agent Systems and Applications III*, pages 394–403, Springer-Verlag.
- [Gomez-Sanz et al., 2002] Gomez-Sanz, J., Pavon, J., and Garijo, F. (2002). Meta-Models for Building Multi-Agent Systems. *Proceedings of the 2002 ACM symposium on Applied computing*, pages 37–41. ACM Press.
- [Graham et al., 1997] Graham, I., Hederson-Sellers, B., and Younessi, H. (1997). *The OPEN Process Specification*. Addison-Wesley.

- [Grassé, 1959] Grassé, P.-P. (1959). La Reconstruction du Nid et les Coordinations Inter-Individuelles chez *Bellicositermes Natalensis* et *Cubitermes* sp. La Théorie de la Stigmergie: Essai d'Interprétation du Comportement des Termites Constructeurs. *Insectes Sociaux*, 6, pages 41–84.
- [Gray and Reuter, 1993] Gray, J., and Reuter, A. (1993). *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann.
- [Gruber, 1991] Gruber, T. R. (1991). The Role of a Common Ontology in Achieving Sharable, Reusable Knowledge Bases. *Proceedings of the Knowledge Representation and Reasoning Conference*, pages 601–602.
- [Guerraoui and Schiper, 1997] Guerraoui, R., and Schiper, A. (1997). Software-Based Replication for Fault Tolerance. *IEEE Computer*, 30(4), pages 68–74.
- [Guessoum and Briot, 1999] Guessoum, Z., and Briot, J.-P. (1999). From Active Objects to Autonomous Agents. *IEEE Concurrency*, 7(3), pages 68–76.
- [Guessoum et al., 2002] Guessoum, Z., Briot, J.-P., and Charpentier, S. (2002). Dynamic and Adaptive Replication for Large-Scale Reliable Multi-Agent Systems. *Proceedings of the 1st International Workshop on Software Engineering for Large-Scale Multi-Agent Systems (SELMAS'02)*, Orlando, USA, ACM Press.
- [Gulyas and Corliss, 1999] Gulyas, T. Kozsik, L., and Corliss, J. B. (1999). The Multi-Agent Modelling Language and the Model Design Interface. *The Journal of Artificial Societies and Social Simulation*, 2(4).
- [Gurevich, 1984] Gurevich, Y. (1984). Toward Logic Tailored for Computational Complexity. *Computation and Proof Theory*, 1104, 175–216.
- [Gustavsson and Fredriksson, 2003] Gustavsson, R., and Fredriksson, M. (2003). Sustainable Information Ecosystems. Garcia, A., Lucena, C, Zambonelli, F., Omicini, A., and Castro, J. (Eds.) *Software Engineering for Large-Scale Multi-Agent Systems*, pp. 127-142, Springer-Verlag.
- [Gutknecht and Ferber, 2001] Gutknecht, O., and Ferber M. (2001). Integrating Tools and Infrastructures for Generic Multi-Agent Systems. *Proceedings of Agents'01*, pages 441-448.
- [Hackwood and Beni, 1991] Hackwood, S., and Beni, G. (1991). Self-Organizing Sensors by Deterministic Annealing. *Proceedings of the IEEE/RSJ International Conference on Intelligent Robot and Systems*, pages 1177–1183.

- [Hackwood and Beni, 1992] Hackwood, S., and Beni, G. (1992). Self-Organization of Sensors for Swarm Intelligence. *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 819–29.
- [Halpern et al., 1995] Halpern, J. Y., Fagin, R., Moses, Y., and Vardi, M. Y. (1995). *Reasoning About Knowledge*. The MIT Press.
- [Hayden et al., 1999] Hayden, S., Carrick, C., and Yang, Q. (1999). Architectural Design Patterns for Multiagent Coordination. *Proceedings of the 3rd International Conference on Autonomous Agents (Agents '99)*, Seattle, USA.
- [Heiler, 1995] Heiler, S. (1995). Semantic Interoperability. *ACM Computing Surveys*, 27(2), pages 271–273.
- [Heusse et al., 1998] Heusse, M., Guérin, S., Snyers, D., and Kuntz, P. (1998). Adaptive Agent-Driven Routing and Load Balancing in Communication Networks. *Advances in Complex Systems*, 1, pages 234–257.
- [Hexmoor, 2001] Hexmoor, H. (2001). A Cognitive Model of Situated Autonomy. *Advances in Artificial Intelligence*, pages 325–334. Springer-Verlag.
- [Hexmoor et al., 2003] Hexmoor, H., Castelfranchi, C., and Falcone, R. (2003). A Prospectus on Agent Autonomy. *Agent Autonomy*, 1(1), pages 1–8.
- [Heylighen, 1992] Heylighen, F. (1992). Evolution, Selfishness and Cooperation; Selfish Memes and the Evolution of Cooperation. *Journal of Ideas*, 2-4, pages 70–84.
- [Hindriks et al., 1999] Hindriks, K. V., Boer, F. S., der Hoek, W. V., and Meyer, J.-J. (1999). Agent Programming in 3APL. *Autonomous Agents and Multi-Agent Systems*, 2(4), pages 357–401.
- [Holland, 1997] Holland, J. H. (1997). *Emergence: From Order to Chaos*. Oxford University Press.
- [Holzmann, 1991] Holzmann, G. J. (1991). *Design and Validation of Computer Protocols*. Prentice-Hall International.
- [Holzmann, 1997] Holzmann, G. J. (1997). The Model Checker Spin. *IEEE Transactions on Software Engineering*, 23(5), pages 279–295.
- [Horn, 2001] Horn, P. M. (2001). Autonomic Computing – IBM’s Perspective on the State of Information Technology. Available at <http://www.ibm.com/research/autonomic>.

- [Hornung and Bryan, 2002] Hornung, E., and Bryan, B. M. (Eds.) (2002). *The Quest for Immortality: Treasures of Ancient Egypt*. National Gallery of Art.
- [Howden et al., 2001] Howden, N., Ronnquist, R., Hodgson, A., and Lucas, A. (2001). JACK Intelligent Agents – Summary of an Agent Infrastructure. *Proceedings of Agents'01*.
- [Huang et al., 1999] Huang, A. C., Ling, B. C., and Ponnkanti, S. (1999). Pervasive Computing: What is it Good For? *Proceedings of The ACM International Workshop on Data Engineering for Wireless and Mobile Access (MOBIDE'1999)*, Seattle, USA.
- [Huberman, 1991] Huberman, B. A. (1991). The Performance of Cooperative Processes. Forrest, S. (Ed.) *Emergent Computation: Self-Organizing, Collective, and Cooperative Phenomena in Natural and Artificial Computing Networks*, The MIT Press.
- [Huget, 2002a] Huget, M.-P. (2002a). Agent UML Class Diagrams Revisited. Bauer, B., Fischer, K., Muller, J., and Rumpe, B. (Eds.) *Proceedings of Agent Technology and Software Engineering (AgeS)*.
- [Huget, 2002b] Huget, M.-P. (2002b). Generating Code for Agent UML Sequence Diagrams. Bauer, B., Fischer, K., Muller, J., and Rumpe, B. (Eds.) *Proceedings of Agent Technology and Software Engineering (AgeS)*.
- [Huget, 2002c] Huget, M.-P. (2002c). Model Checking Agent UML Protocol Diagrams. *Proceedings of the Workshop on Model Checking Artificial Intelligence (MoChArt)*.
- [Huget, 2002d] Huget, M.-P. (2002d). Nemo: An Agent-Oriented Software Engineering Methodology. *Proceedings of the Workshop on Agent-Oriented Methodologies*, pages 41–53, Seattle, USA.
- [Huhns and Singh, 1998] Huhns, M. N., and Singh, M. P. (1998). Agents and Multiagent Systems: Themes, Approaches, and Challenges. Huhns, M. N., and Singh, M. P. (Eds.) *Reading in Agents*, pages 1–23.
- [Huhns and Singh, 1999] Huhns, M. N., and Singh, M. P. (1999). A Multiagent Treatment of Agenthood. *Applied Artificial Intelligence*, 13(1–2), pages 3–10.
- [Huhns and Stephens, 1999] Huhns, M. N., and Stephens, L. M. (1999). Multiagent Systems and Societies of Agents. Weiss, G. (Ed.) *Multiagent Systems*, pages 79–120, The MIT Press.

- [Huhns et al., 2002] Huhns, M. N., Stephens, L. M., and Ivezic, N. (2002). Automating Supply-Chain Management. *Proceedings of the 1st International Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*, pages 1017–1024, ACM Press.
- [Huzita and Scimemi, 1989] Huzita, H., and Scimemi, B. (1989). The Algebra of Paper-Folding. *Proceedings of the 1st International Meeting of Origami Science and Technology*.
- [Iglesias, 1998] Iglesias, C. (1998). *Definicion de una Metodologia para el Desarrollo de Sistemas Multi-Agente*. Ph.D. Thesis, Universidad Politecnica de Madrid.
- [Iglesias et al., 1998a] Iglesias, C. A., Garrijo, M., González, J. (1998a). A Survey of Agent-Oriented Methodologies. *Agent Theories, Architectures and Languages*.
- [Iglesias et al., 1998b] Iglesias, C. A., Garijo, M., Gonzales, J. C., and Velasco, J. R. (1998b). Analysis and Design of Multi-Agent Systems Using MAS-CommonKADS. Singh, M. P., Rao, A., and Wooldridge, M. J. (Eds.) *Intelligent Agents IV*, pages 313–326. Springer-Verlag.
- [Inbody, 2003] Inbody, D. (2003). Swarming: Historical Observations and Conclusions. *Proceedings of Swarming: Network Enabled C4ISR*, Tysons Corner, VA.
- [Intanagonwiwat et al., 2000] Intanagonwiwat, C., Govindan, R., Estrin, D. (2000). Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensor Networks. *Proceedings of MobiCom 2000*.
- [d’Inverno and Luck, 1996] d’Inverno, M., and Luck, M. (1996). A Formal View of Social Dependence Networks. *Proceedings of the 1st Australian Workshop on Distributed Artificial Intelligence*, pages 115–129, Springer-Verlag.
- [d’Inverno et al., 2000] d’Inverno, M., Hindriks, K., and Luck, M. (2000). A Formal Architecture for the 3APL Agent Programming Language. *Proceedings of the 1st International Conference of B and Z Users*, pages 168–187, Springer-Verlag.
- [ITU, 1999] International Telecommunications Union (1999). Languages for Telecommunication Applications, Formal description techniques (FDT): Message Sequence Charts (MSC). International Telecommunications Union.
- [Jacob, 2001] Jacob, C. (2001). *Illustrating Evolutionary Computation with Mathematica*. Morgan Kaufmann.

- [Jacobson et al., 1999] Jacobson, I., Booch, G., and Rumbaugh, J. (1999). *The Unified Software Development Process*. Addison-Wesley.
- [Jeng and Cheng, 1995] Jeng, J.-J., Cheng, B. H. C. (1995). Specification Matching for Software Reuse: A Foundation. Proceedings of the ACM SIGSOFT Symposium Software Reusability, ACM Press.
- [Jennings, 2000] Jennings, N. R. (2000). On Agent-Based Software Engineering. *Artificial Intelligence*, 117(2), pages 277–296.
- [Jennings, 2001] Jennings, N. R. (2001). An Agent-Based Approach for Building Complex Software Systems. *Communications of the ACM*, 44(4), pages 35–41.
- [Jennings et al., 1998] Jennings, N. R., Sycara, K., and Wooldridge, M. J. (1998). A Roadmap of Agent Research and Development. *Autonomous Agents and Multi-Agent Systems*, 1(1).
- [Jennings et al., 2000] Jennings, N. R., Faratin, P., Norman, T. J., O'Brien, P., and Odgers, B. (2000). Autonomous Agents for Business Process Management. *International Journal of Applied Artificial Intelligence*, 14(2).
- [Johansson and Saffiotti, 2001] Johansson, S. J., and Saffiotti, A. (2001). Using the Electric Field Approach in the RoboCup Domain. *Proceedings of RoboCup*.
- [Johnson and Zweig, 1991] Johnson R. E., Zweig, J. M. (1991). Delegation in C++. *The Journal of Object Oriented Programming*, 4(7), pages 31–34.
- [Joyce et al., 1987] Joyce, J., Lomow, G., Slind, K., and Unger, B. (1987). Monitoring Distributed Systems. *ACM Transactions on Computer Systems*, 5(2), pages 121–150.
- [Juan et al., 2002] Juan, T., Pierce, A., and Sterling, L. (2002). ROADMAP: Extending the Gaia methodology for Complex Open Systems. *Proceedings of the 1st ACM Joint Conference on Autonomous Agents and Multi-Agent Systems*, pages 3–10, ACM Press.
- [Judge et al., 1998] Judge, D. W., Odgers, B. R., Shepherdson, J. W., and Cui, Z. (1998). Agent-Enhanced Workflow. *BT Technology Journal*, 16(3), pages 79–85.
- [Julian and Botti, 2004] Julian, V., and Botti, V. (2004) Developing Real-Time Multi-Agent Systems. To appear in *Integrated Computer Aided Engineering Journal*, 2004.

- [Kagal et al., 2001] Kagal, L., Finin, T., and Joshi, A. (2001). Moving from Security to Distributed Trust in Ubiquitous Computing Environments. *IEEE Computer*, 34(12).
- [Kahn and Cicalese, 2001] Kahn, M. L., and Cicalese, C. (2001). CoABS Grid Scalability Experiments. *Proceedings of the 2nd International Workshop on Infrastructure for Scalable Multi-Agent systems at Autonomous Agents*, Montreal, CA.
- [Kahn et al., 1999] Kahn, J. M., Katz, R. H., and Pister, K. S. (1999). Mobile Networking for Smart Dust. *Proceedings of MobiCom'99*.
- [Karamcheti et al., 1996] Karamcheti, V., Plevyak, J., and Chien, A., (1996). Runtime Mechanisms for Efficient Dynamic Multithreading. *Journal of Parallel and Distributed Computing*, 37, pages 21–40.
- [Kawamura et al., 1999] Kawamura, T., Yoshioka, N., Hasegawa, T., Ohsuga, A., and Honiden, S., (1999). Bee-gent: Bonding and Encapsulation Enhancement Agent Framework for Development of Distributed Systems. *Proceedings of the 6th Asia-Pacific Software Engineering Conference*.
- [Kendall, 1998] Kendall, E. A. (1998). Agent Roles and Role Models: New Abstractions for Multiagent System Analysis and Design. *Proceedings of the International Workshop on Intelligent Agents in Information and Process Management*.
- [Kendall, 2000] Kendall, E. A. (2000). Agent Software Engineering with Role Modelling. *Proceedings of the Workshop on Agent-Oriented Software Engineering*, pages 163–169, Springer-Verlag.
- [Kendall and Malkoun, 1996] Kendall, E. A., and Malkoun, M. T. (1996). The Layered Agent Patterns. *Pattern Languages of Programs (PLoP'96)*.
- [Kendall et al., 1995] Kendall, E. A., Malkoun, M. T., and Jiang, C. H. (1995). A Methodology for Developing Agent Based Systems. Zhang, C., and Lukose, D. (Eds.) *Proceedings of the 1st Australian Workshop on Distributed Artificial Intelligence*.
- [Kennedy et al., 2001] Kennedy, J., Eberhart, R. C., and Shi, Y. (2001). *Swarm Intelligence*. Morgan Kaufmann.
- [Kephart, 2002] Kephart, J. (2002). Software Agents and the Route to the Information Economy. *Proceedings of the National Academy of Science*, 99(3), pages 7207–7213.
- [Kerr et al., 1998] Kerr, D., O'Sullivan, D., Evans, R., Richardson, R., and Somers, F., (1998). Experiences using Intelligent Agent Technologies as

- a Unifying Approach to Network and Service Management. *Proceedings of IS & N'98*, Antwerp, Belgium.
- [Kiczales et al., 1991] Kiczales, G., des Rivières, J., and Bobrow, D. G. (1991). *The Art of the Metaobject Protocol*. The MIT Press.
- [Kim, 1997] Kim, J. (1997). Explanation, Prediction, and Reduction in Emergentism. *Intellectica – Emergence and Explanation*, 2-25.
- [Kinny and Georgeff, 1996] Kinny, D., and Georgeff, M. (1996). Modelling and Design of Multi-Agent Systems. *Intelligent Agents III*, Springer-Verlag.
- [Kinny et al., 1996] Kinny, D., Georgeff, M., and Rao, A. (1996). A Methodology and Modelling Technique for Systems of BDI Agents. van der Velde, W., and Perram, J. (Eds.) *Agents Breaking Away*, pages 56–71. Springer-Verlag.
- [Klusck and Sycara, 2001] Klusck, M., and Sycara, K. (2001). Brokering and Matchmaking for Coordination of Agent Societies: A Survey. Omicini, A., Zambonelli, F., Klusck, M., and Tolksdorf, R. (Eds.) *Coordination of Internet Agents: Models, Technologies, and Applications*, pages 197–224.
- [Knublauch and Rose, 2002] Knublauch, H. H., and Rose, T. (2002). Tool-Supported Process Analysis and Design for the Development of Multi-Agent Systems. *Proceedings of AOSE 2002*, Springer-Verlag.
- [Kolp et al., 2001] Kolp, M., Giorgini, P., and Mylopoulos, J. (2001). A Goal-Based Organizational Perspective on Multi-Agents Architectures. *Proceedings of the 8th International Workshop on Agent Theories, Architectures, and Languages, ATAL'01*, Seattle, USA.
- [Koning and Romero-Hernandez, 2002] Koning, J.-L., and Romero-Hernandez, I. (2002). Generating Machine Processable Representations of Textual Representations of AUML. Giunchiglia, F, Odell, J., and Weiss, G. (Eds.) *Proceedings of the Workshop on Agent-Oriented Software Engineering (AOSE)*.
- [Kortuem and Segall, 2003] Kortuem, G., and Segall, Z. (2003). Wearable Communities: Augmenting Social Networks with Wearable Computers. *IEEE Pervasive Computing Magazine*, 2(1).
- [Kouadri Mostéfaoui, 2003] Kouadri Mostéfaoui, S. (2003). Towards a Context-Oriented Services Discovery and Composition Framework. *Proceedings of the Workshop on Artificial Intelligence, Information Access, and Mobile Computing, Acapulco, Mexico*.

- [Kruchten, 2000] Kruchten, P. (2000). *The Rational Unified Process: An Introduction*. Addison-Wesley.
- [Kumar, 2002] Kumar, M. (2002). Contrast and Comparison of Five Major Agent Oriented Software Engineering (AOSE) methodologies. Available at <http://students.jmc.ksu.edu/grad/madhukar/www/professional/aosepaper.pdf>.
- [Kuutti, 1991] Kuutti, K. (1991). The Concept of Activity as a Basic Unit of Analysis for CSCW Research. *Proceedings of the 2nd European Conference on Computer Supported Cooperative Work (ECSCW '91)*, pages 249–264, Kluwer Academic Publishers.
- [Kuwabara et al., 1996] Kuwabara, K., Ishida, T., Nishibe, Y., and Suda, T. (1996). An Equilibratory Market-Based Approach for Distributed Resource Allocation and Its Applications to Communication Network Control. *Market-Based Control: A Paradigm for Distributed Resource Allocation*, pages 53–73, World Scientific Publishing.
- [Labrou and Finin, 1997] Labrou, Y., and Finin, T. (1997). A Proposal for a New KQML Specification. Technical Report TR-CS-97-03, University of Maryland.
- [Labrou et al., 1999] Labrou, Y., Finin, T., and Peng, Y. (1999). The Current Landscape of Agent Communication Languages. *Intelligent Systems*, 14(2), pages 45–52.
- [Lacey et al., 2000] Lacey, T. H., and DeLoach, S. A. (2000). Automatic Verification of Multiagent Conversations. *Proceedings of the 11th Annual Midwest Artificial Intelligence and Cognitive Science Conference*, Fayetteville, USA.
- [Laird et al., 1987] Laird, J. E., Newell, A., and Rosenbloom, P. S. (1987). SOAR: An Architecture for General Intelligence. *Artificial Intelligence*, 33(1), pages 1–64.
- [Lange and Oshima, 1999] Lange, D., and Oshima, M. (1999). Dispatch Your Agents; Shut Off Your Machine. *Communications of the ACM*, 42(3).
- [Lavender and Schmidt, 1996] Lavender, G., and Schmidt, D., (1996). Active Object: An Object Behavioural Pattern for Concurrent Programming. Vlissides, J.M., Coplien, J.O., and Kerth, N.L. (Eds.) *Pattern Languages of Program Design*, Addison-Wesley.
- [Lawley et al., 2003] Lawley, R., Decker, K., Luck, M., Payne, T., and Moreau, L. (2003). Automated Negotiation for Grid Notification Services.

- Proceedings of the 9th International Europar Conference (EURO-PAR'03)*, Springer-Verlag.
- [Lawrence, 1992] Lawrence, P. A. (1992). *The Making of a Fly: the Genetics of Animal Design*. Blackwell Science.
- [Lea, 1997] Lea, D. (1997). *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley.
- [Leontjev, 1978] Leontjev, A. (1978). *Activity, Consciousness, and Personality*. Prentice-Hall International.
- [Lesperance et al., 1999] Lesperance, Y., Kelley, T. G., Mylopoulos, J., and Yu, E. (1999). Modeling Dynamic Domains with Congolog. *Proceedings of CAiSE'99*, pages 365–380, Springer-Verlag.
- [Leyman, 2001] Leyman, F. (2001). *Web Services Flow Language (WSFL)*. Technical report, IBM.
- [Lind, 2000] Lind, J. (2000). A Development Method for Multiagent systems. *Proceedings of the 15th European Meeting on Cybernetics and Systems Research*.
- [Lind, 2001] Lind, J. (2001). *Iterative Software Engineering for Multi-Agent Systems, The MASSIVE Method*. Springer-Verlag.
- [Loo, 2003] Loo, A. W. (2003). The Future of Peer-to-Peer Computing. *Communications of the ACM*, 46(9), pages 56–61, ACM Press.
- [Lord et al., 2003] Lord, P., Wroe, C., Stevens, R., Goble, C., Miles, S., Moreau, L., Decker, K., Payne, T., and Papay, J. (2003). Semantic and Personalised Service Discovery. Cheung, W. K., and Ye, Y. (Eds.) *Proceedings of the Workshop on Knowledge Grid and Grid Intelligence (KGGI'03)*, pages 100–107.
- [Luck and d'Inverno, 1995] Luck, M., and d'Inverno, M. (1995). A Formal Framework for Agency and Autonomy. Lesser, V., and Gasser, L. (Eds.) *Proceedings of the 1st International Conference on Multi-Agent Systems (ICMAS-95)*, pages 254–260, AAAI Press.
- [Luck and d'Inverno, 2001] Luck, M., and d'Inverno, M. (2001). *A Conceptual Framework for Agent Definition and Development*. *The Computer Journal* 44(1), pages 1–20.
- [Luck et al., 2003] Luck, M., McBurney, P., and Preist, C. (2003). *Agent Technology: Enabling Next Generation Computing*. AgentLink II.

- [Lynch, 1996] Lynch, N. (1996). *Distributed Algorithms*. Morgan Kaufmann.
- [Lyytinen and Yoo, 2002] Lyytinen, K., and Yoo, Y. (2002). Issues and Challenges in Ubiquitous Computing. *Communications of the ACM*, 45(12).
- [Maamar et al., 2001] Maamar, Z., Dorion, E., and Daigle, C. (2001). Towards Virtual Marketplaces for E-Commerce. *Communications of the ACM*, 44(12).
- [Maamar et al., 2004] Maamar, Z., Sheng, Q. Z., and Benatallah, B. (2004). On Composite Web Services Provisioning in an Environment of Fixed and Mobile Computing Resources. *Information Technology and Management Journal*, 5(3), Kluwer Academic Publishers.
- [Machado, 2003] Machado, R. (2003). SIM Speak. Available at http://www.inf.ufrgs.br/~bordini/SIM_Speak.
- [Machado and Bordini, 2001] Machado, R., and Bordini, R. H. (2001). Running AgentSpeak(L) Agents on SIM_AGENT. Meyer, J. J., and Tambe, M. (Eds.) *Intelligent Agents VIII*, pages 158–174.
- [Maes, 1994] Maes, P. (1994). Agents that Reduce Work and Information Overload. *Communications of the ACM*, 37(7), pages 31–40.
- [Malone and Crowston, 1994] Malone, T., and Crowston, K. (1994). The Interdisciplinary Study of Coordination. *ACM Computing Surveys*, 26(1), pages 87–119.
- [Mamei et al., 2003a] Mamei, M., Zambonelli, F., and Leonardi, L. (2003a). Tuples on the Air: a Middleware for Context-Aware Computing in Dynamic Networks. *Proceedings of the International ICDCS Workshop on Mobile Computing Middleware*.
- [Mamei et al., 2003b] Mamei, M., Zambonelli, F., and Leonardi, L. (2003b). Distributed Motion Coordination with Co-Fields: A Case Study in Urban Traffic Management. *Proceedings of the IEEE Symposium on Autonomous Decentralized Systems*.
- [Mangina, 2002] Mangina, E. (2002). Review of Software Products for Multi-Agent Systems. AgentLink II.
- [Marik et al., 2003] Marik, V., Pechoucek, M., Vrba, P., and Hrdonka, V., (2003). FIPA Standards and Holonic Manufacturing. Agent Based Manufacturing. Deen, S. M. (Ed.), *Advances in the Holonic Approach*, pages 89–121, Springer-Verlag.

- [Massonet et al., 2002] Massonet, P., Deville, Y., and Neve, C. (2002). From AOSE Methodology to Agent Implementation. *Proceedings of the 1st International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'02)*, Bologna, Italy.
- [Mathieson et al., 2004] Mathieson, I., Dance, S., Padgham, L., Gorman, M., and Winikoff, M. (2004). An Open Meteorological Alerting System: Issues and Solutions. *Proceedings of the 27th Australasian Computer Science Conference*, Dunedin, New Zealand.
- [Maximilien and Singh, 2002] Maximilien, E. M., and Singh, M. P. (2002). Reputation and Endorsement for Web Services. *ACM SIGEcom Exchanges*, 3(1), pages 24–31.
- [McCabe and Clark, 1995] McCabe, F. G., and Clark, K. L. (1995). April – Agent PProcess Interaction Language. Wooldridge, M. J., and Jennings, N. (Eds.) *Intelligent Agents*, Springer-Verlag.
- [McCarthy, 1978] McCarthy, J. (1978). History of LISP. *The 1st ACM SIGPLAN Conference on History of Programming Languages*, pages 217–223.
- [McCarthy and Hayes, 1981] McCarthy, J., and Hayes, P. J. (1981). Some Philosophical Problems from the Standpoint of Artificial Intelligence. Weber, B. L. and Nilsson, N. J. (Eds.) *Readings in Artificial Intelligence*, pages 431–450. Morgan Kaufmann.
- [McIlraith and Martin, 2003] McIlraith, S., and Martin, D. (2003). Bringing Semantics to Web Services. *IEEE Intelligent Systems*, 18(1), pages 90–93.
- [Medvidovic and Taylor, 1997] Medvidovic, N., and Taylor, R. N. (1997). A Framework for Classifying and Comparing Architecture Description Languages. *Proceedings of the 5th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 60–76, Springer-Verlag.
- [Meling et al., 2001] Meling, H., Montresor, A., and Babaoglu, O. (2001). Peer-to-Peer Document Sharing using the Ant Paradigm. *Proceedings of the Norsk Informatikkonferanse (NIK)*.
- [Mendez and Tolksdorf, 2003] Mendez, R., and Tolksdorf R. (2003). A New Approach to Scalable Linda-Systems Based on Swarms. *Proceedings of ACM SAC '03*.
- [Messer et al., 2002] Messer, A., Greeberg, I., Bernadat, P., and Milojicic, D. (2002). Towards a Distributed Platform for Resource-Constrained Devices. *Proceedings of the IEEE 22nd International Conference on Distributed Computing Systems (ICDCS'2002)*, Vienna, Austria.

- [Meyer, 1992] Meyer, B. (1992). Applying Design by Contract. *IEEE Computer*, 25(10), pages 40–51.
- [Meyer, 1997] Meyer, B. (1997). *Object-Oriented Software Construction*. Prentice-Hall International.
- [Miles et al., 2000] Miles, S., Joy, M., and Luck, M. (2000). Designing Agent-Oriented Systems by Analysing Agent Interactions. Ciancarini, P., and Wooldridge, M. J. (Eds.) *Proceedings of Agent-Oriented Software Engineering 2000 (AOSE 2000)*, pages 171–184.
- [Miles et al., 2003] Miles, S., Papay, J., Dialani, V., Luck, M., Decker, K., Payne, T., and Moreau, L. (2003). Personalised Grid Service Discovery. *IEE Proceedings of Software: Special Issue on Performance Engineering*, 150(4), pages 252–256.
- [Miller et al., 1996] Miller, M. S., Krieger, D., Hardy, N., Hibbert, C., and Tribble, D. E. (1996). An Automated Auction in ATM Network Bandwidth. *Market-Based Control. A Paradigm for Distributed Resource Allocation*, pages 96–125.
- [Milojicic et al., 1998] Milojicic, D., Breugst, M., Busse, I., Campbell, J., Covaci, S., Friedman, B., Kosaka, K., Lange, D., Ono, K., Oshima, M., Tham, C., Virdhagriswaran, S., and White J. (1998). MASIF – The OMG Mobile Agent System Interoperability Facility. Rothermel K., and Hohl, F. (Eds.) *Proceedings of the 2nd International Workshop Mobile Agents*, pages 50–67, Springer-Verlag.
- [Mintzberg, 1992] Mintzberg, H. (1992). *Structure in Fives: Designing Effective Organizations*. Prentice-Hall International.
- [Morabito et al., 1999] Morabito, J., Sack, I., and Bhate, A. (1999). *Organization Modeling: Innovative Architectures for the 21st Century*. Prentice-Hall International.
- [Moraitis et al., 2002] Moraitis, P., Petraki, E., and Spanoudakis, N. I. (2002). Engineering JADE Agents with the Gaia Methodology. *Proceedings of the International Workshop on Agents and Software Engineering*.
- [Moreau, 2002] Moreau, L. (2002). Agents for the Grid: A Comparison with Web Services (Part 1: The Transport Layer). Bal, H. E., Lohr, K.-P., and Reinefeld, A. (Eds.) *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID 2002)*, pages 220–228, Berlin, Germany.
- [Moreau et al., 2003] Moreau, L., Miles, S., Goble, C., Greenwood, M., Dialani, V., Addis, M., Alpdemir, N., Cawley, R., De Roure, D., Ferris, J.,

- Gaizauskas, R., Glover, K., Greenhalgh, C., Li, P., Liu, X., Lord, P., Luck, M., Marvin, D., Oinn, T., Paton, N., Pettifer, S., Radenkovic, M. V., Roberts, A., Robinson, A., Rodden, T., Senger, M., Sharman, N., Stevens, R., Warboys, B., Wipat, A., and Wroe, C. (2003). On the Use of Agents in a BioInformatics Grid. *Proceedings of the Third IEEE/ACM CCGRID'2003 Workshop on Agent Based Cluster and Grid Computing*, pages 653–661, Tokyo, Japan.
- [Moss, 2000] Moss, S. (2000). *Editorial Introduction: Messy Systems – The Target for Multi Agent Based Simulation*, Springer-Verlag.
- [Moulin and Chaib-draa, 1996] Moulin, B., and Chaib-draa, B. (1996). An Overview of Distributed Artificial Intelligence. *Foundations of Distributed Artificial Intelligence*, John Wiley & Sons.
- [Muller, 1996] Muller, J. P. (1996). *The Design of Intelligent Agents: A Layered Approach*, Springer-Verlag.
- [Muller, 2003] Muller, J. P. (2003). The Right Agent (Architecture) to do the Right Thing. *Intelligent Agents V*, pages 105–112, Springer-Verlag.
- [Muller and Pischel, 1994] Muller, J., and Pischel, M. (1994). Modelling Reactive Behaviour in Vertically Layered Agent Architectures. Cohen, A. G. (Ed.) *11th European Conference on Artificial Intelligence (ECAI'94)*, pages 709–713.
- [Mylopoulos and Castro, 2000] Mylopoulos, J., and Castro, J. (2000). Tropos: A Framework for Requirements-Driven Software Development. *Proceedings of 12th Conference on Advanced Information Systems Engineering (CAISE)*.
- [Nakajima et al., 2003] Nakajima, T., Awa, I., and Tokunaga, E. (2003). A Proactive Middleware Platform for Mobile Computing. *Proceedings of the 4th ACM International Middleware Conference (Middleware'2003)*, Rio de Janeiro, Brazil.
- [Nagpal, 2001] Nagpal, 2001. Programmable Self-Assembly: Constructing Global Shape using Biologically-Inspired Local Interactions and Origami Mathematics. Ph.D. Thesis, MIT, 2001.
- [Nagpal, 2002] Nagpal, R. (2002). Programmable Self-Assembly Using Biologically-Inspired Multiagent Control. *Proceedings of the Conference on Autonomous Agents and Multiagent Systems (AAMAS)*.
- [Nagpal and Coore, 1998] Nagpal, R., and Coore, D. (1998). An Algorithm for Group Formation in an Amorphous Computer. *Proceedings of the In-*

- ternational Conference on Parallel and Distributed Computing and Systems (PDCS).*
- [Nagpal et al., 2003] Nagpal, R., Shrobe, H., and Bachrach, J. (2003). Organizing a Global Coordinate System from Local Information on an Ad Hoc Sensor Network. *Proceedings of the 2nd International Workshop on Information Processing in Sensor Networks (IPSN).*
- [Ndumu et al., 1999] Ndumu, D. T., Nwana, H. S., Lee, L. C., and Collis, J. C. (1999). Visualising and Debugging Distributed Multi-Agent Systems. *Proceedings of the 3rd Annual Conference on Autonomous Agents*, pages 326–333, ACM Press.
- [Neufeld, 1980] Neufeld, E. (1980). Insects as Warfare Agents in the Ancient Near East. *Orientalia*, 49(1), pages 30–57.
- [Newell, 1982] Newell, A. (1982). The Knowledge Level. *Artificial Intelligence*, 7(18), pages 87–127.
- [Newell, 1993] Newell, A. (1993). Reflections on the Knowledge Level. *Artificial Intelligence*, 59, pages 31–38.
- [Nilsson, 1971] Nilsson, N. (1971). *Problem Solving Methods in Artificial Intelligence*. McGraw Hill.
- [Noriega and Sierra, 1999] Noriega, P., and Sierra, C. (1999). Auctions and Multi-agent Systems. Klusch, M. (Ed.) *Information Agents*, pages 153–175, Springer-Verlag.
- [Noriega and Sierra, 2002] Noriega, P., and Sierra, C. (2002). Electronic Institutions: Future Trends and Challenges. Klusch, M., Ossowski, S., and Shehory, O. (Eds.) *Cooperative Information Agents VI*, Springer-Verlag.
- [Nuseibeh and Easterbrook, 2000] Nuseibeh, B. A., and Easterbrook, S. M. (2000). Requirements Engineering: A Roadmap. *Proceedings of the 22nd International Conference on Software Engineering (ICSE'00)*, pages 35–46.
- [Nutt, 1996] Nutt, G. (1996). The Evolution Toward Flexible Workflow Systems. *Distributed Systems Engineering*, 3(4), pages 276–294.
- [Nwana, 1996] Nwana, H. S. (1996). Software Agents: An Overview. *The Knowledge Engineering Review*, 11(3), pages 205–244.
- [Nwana and Ndumu, 1999] Nwana, H. S., and Ndumu, D. T. (1999). A Perspective on Software Agents Research. *The Knowledge Engineering Review*, 14(2), pages 1–18.

- [Nwana et al., 1999] Nwana, H. S., Ndumu, D. T., Lee, L. C., and Collis, J. C. (1999). Zeus: A Toolkit for Building Distributed Multi-agent Systems. *Applied Artificial Intelligence Journal*, 1(13), pages 129–185.
- [Oaks and Wong, 2000] Oaks, S., and Wong, H. (2000). *Jini in a Nutshell*. O'Reilly.
- [Obreiter et al., 2003] Obreiter, P., Konig-Ries, B., and Klein, M. (2003). Stimulating Cooperative Behavior of Autonomous Devices – An Analysis of Requirements and Existing Approaches. Technical Report 2003-1, University of Karlsruhe.
- [Odell, 2002] Odell, J. (2002). Objects and Agents Compared. *Journal of Object Computing*, 1:1.
- [Odell et al., 2000] Odell, J., Parunak, V. H., and Bauer, B. (2000). Extending UML for Agents. *Proceedings of the Agent Oriented Information Systems (AOIS) Workshop*.
- [Odell et al., 2001] Odell, J., Parunak, V. H., and Bauer, B. (2001). Representing Agent Interaction Protocols in UML. Ciancarini, P., and Wooldridge, M. J. (Eds.) *Proceedings of the First International Workshop on Agent Oriented Software Engineering (AOSE-2000)*, pages 121–140, Springer-Verlag.
- [OMG, 1999] Object Management Group (1999). Mobile Agent System Interoperability Facility (MASIF). Available at <http://www.fokus.gmd.de/research/cc/ecco/masif>.
- [OMG, 2000a] Object Management Group (2000a). CORBA 2.4.2 Specification. Available at <http://www.omg.org>.
- [OMG, 2000b] Object Management Group (2000b). Meta Object Facility (MOF). Available at <http://www.omg.org>.
- [OMG, 2000c] Object Management Group (2000c). Unified Modeling Language Specification. Version 1.3. Available at <http://www.omg.org>.
- [OMG, 2001] Object Management Group (2001). Model Driven Architecture (MDA). Technical report, OMG.
- [OMG, 2002] Object Management Group (2002). Software Process Engineering Metamodel. Version 1.0. Available at <http://www.omg.org/technology/documents/formal/spem.htm>.
- [OMG, 2003a] Object Management Group (2003a). XML Metadata Interchange – Version 1.1. Available at <http://www.omg.org>.

- [OMG, 2003b] Object Management Group (2003b). Unified Modeling Language: Superstructure – Version 2.0. Available at <http://www.omg.org>.
- [OMG, 2003c] Object Management Group (2003c). Response to the UML 2.0 OCL RfP. Available at <http://www.omg.org>.
- [Omicini, 2001] Omicini, A. (2001). SODA: Societies and Infrastructures in the Analysis and Design of Agent-Based Systems. Ciancarini, P., and Wooldridge, M. J. (Eds.) *Agent-Oriented Software Engineering*, pages 185–193. Springer-Verlag.
- [Omicini, 2002] Omicini, A. (2002). Towards a Notion of Agent Coordination Context. Marinescu, D. C., and Lee, C. (Eds.) *Process Coordination and Ubiquitous Computing*, pages 187–200. CRC Press.
- [Omicini and Denti, 2001] Omicini, A., and Denti, E. (2001). From Tuple Spaces to Tuple Centres. *Science of Computer Programming*, 41(3), pages 277–294.
- [Omicini and Ossowski, 2003] Omicini, A., and Ossowski, S. (2003). Objective versus Subjective Coordination in the Engineering of Agent Systems. Klusch, M., Bergamaschi, S., Edwards, P., and Petta, P. (Eds.) *Intelligent Information Agents: An AgentLink Perspective*, pages 179–202, Springer-Verlag.
- [Omicini and Ricci, 2003] Omicini, A., and Ricci, A. (2003). Reasoning about Organisation: Shaping the Infrastructure. *AI*IA Notizie*, XVI(2), pages 7–16.
- [Omicini and Zambonelli, 1999] Omicini, A., and Zambonelli, F. (1999). Coordination for Internet Application Development. *International Journal on Autonomous Agents and Multi-Agent Systems*, 2(3), pages 251–269.
- [Omicini et al., 2003] Omicini, A., Ricci, A., and Viroli, M. (2003). Formal Specification and Enactment of Security Policies through Agent Coordination Contexts. Focardi, R., and Zavattaro, G. (Eds.) *Security Issues in Coordination Models, Languages and Systems*, Elsevier Science.
- [Ossowski, 1999] Ossowski, S. (1999). *Coordination in Artificial Agent Societies – Social Structure and Its Implications for Autonomus Problem-solving Agents*, Springer-Verlag.
- [Ossowski and Omicini, 2002] Ossowski, S., and Omicini, A. (2002). Coordination Knowledge Engineering. *The Knowledge Engineering Review*, 17(4), pages 309–316.

- [Ossowski et al., 2002] Ossowski, S., Hernández, J. Z., Iglesias, C. A., and Fernández, A. (2002). Engineering Agent Systems for Decision Support. Petta, P., Tolksdorf, R., and Zambonelli, F. (Eds.) *Engineering Societies in an Agent World III*. Springer-Verlag.
- [Ott et al., 1990] Ott, E., Grebogi, C., and Yorke, J. A. (1990). Controlling Chaos. *Physical Review Letters*, 64(11), pages 1196–1199.
- [Padgham and Winikoff, 2002] Padgham, L., and Winikoff, M. (2002). Prometheus: A Pragmatic Methodology for Engineering Intelligent Agents. *Proceedings of the Workshop on Agent-Oriented Methodologies*, pages 97–108.
- [Paolucci et al., 2002] Paolucci, M., Kawamura, T., Payne, T. R., and Sycara, K. (2002). Importing the Semantic Web in UDDI. *Proceedings of the Workshop on Web Services, E-Business and Semantic Web*.
- [Papadopoulos and Arbab, 1998] Papadopoulos, G. A., and Arbab, F. (1998). Coordination Models and Languages. *The Engineering of Large Systems*, 46.
- [Papasimeon and Heinze, 2001] Papasimeon, M., and Heinze, C. (2001). Extending the UML for Designing JACK Agents. *Proceedings of the Australian Software Engineering Conference (ASWEC 01)*.
- [Parunak, 1990] Parunak, H. V. D. (1990). Distributed AI and Manufacturing Control: Some Issues and Insights. Demazeau, Y., and Muller, J.-P. (Eds.) *Decentralized AI*, pages 81–104.
- [Parunak, 1997] Parunak, H.V.D. (1997). Go to the Ant: Engineering Principles from Natural Agent Systems. *Annals of Operations Research*, 75, pages 69–101.
- [Parunak and Brueckner, 2001] Parunak, H. V. D., and Brueckner, S. (2001). Entropy and Self-Organization in Multi-Agent Systems. *Proceedings of the 5th International Conference on Autonomous Agents (Agents 2001)*, pages 124–130, ACM Press.
- [Parunak and Brueckner, 2003] Parunak, H. V. D., and Brueckner, S. (2003). Swarming Coordination of Multiple UAV's for Collaborative Sensing. *Proceedings of 2nd AIAA "Unmanned Unlimited" Systems, Technologies, and Operations Conference*, San Diego, USA.
- [Parunak and Odell, 2001] Parunak, H. V. D., and Odell, J. (2001). Representing Social Structures in UML. *Proceedings of the 5th International Conference on Autonomous Agents*, pages 100–101. ACM Press.

- [Parunak and Vanderbok, 1997] Parunak, H. V. D., and Vanderbok, R. S. (1997). Managing Emergent Behavior in Distributed Control Systems. *Proceedings of ISA-Tech'97*.
- [Parunak et al., 1998a] Parunak, H. V. D., Savit, R., and Riolo R. L. (1998). Agent-Based Modeling vs. Equation-Based Modeling: A Case Study and Users' Guide. *Proceedings of the Workshop on Modeling Agent Based Systems*, pages 1–15.
- [Parunak et al., 1998b] Parunak, H. V. D., Sauter, J., and Clask, S. (1998). Toward the Specification and Design of Industrial Synthetic Ecosystems. Singh, M., Rao, A., and Wooldridge, M. J. (Eds.) *Agent Theories, Architectures, and Languages (ATAL)*, pp. 45–59, Springer-Verlag.
- [Parunak et al., 2000] Parunak, H. V. D., Brueckener, S. A., Sauter, J., and Matthews, R. (2000). Distinguishing Environmental Properties and Agent Dynamics: A Case Study in Abstraction and Alternate Modelling Technologies. Omicini, A., Tolksdorf, R., and Zambonelli, F. (Eds.) *Engineering Societies in the Agents' Eorld (ESAW)*, pp. 19–33, Springer-Verlag.
- [Parunak et al., 2002a] Parunak, H. V. D., Brueckner, S. A., and Sauter, J. (2002a). Digital Pheromone Mechanisms for Coordination of Unmanned Vehicles. *Proceedings of 1st International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS 2002)*, pages 449–450.
- [Parunak et al., 2002b] Parunak, H. V. D., Purcell, M., and O'Connell, R. (2002b). Digital Pheromones for Autonomous Coordination of Swarming UAV's. *Proceedings of 1st AIAA "Unmanned Aerospace Vehicles" Systems, Technologies, and Operations Conference*, Norfolk, VA.
- [Pasquier and Chaib-draa, 2003] Pasquier, P., and Chaib-draa, B. (2003). The Cognitive Approach for Agent Communication Pragmatics. *Proceedings of the 2nd International Joint Conference on Autonomous Agents and Multi-agent Systems*, pages 544–551, ACM Press.
- [Patil et al., 1992] Patil, R. S., Fikes, R. E., Patel-Schneider, P. F., McKay, D., Finin, T., Gruber, T., and Neches, R. (1992). The DARPA Knowledge Sharing Effort: Progress Report. *Proceedings of the 3rd Conference on Principles of Knowledge Representation and Reasoning*, pages 103–114. Cambridge, USA.
- [Pavón and Gómez-Sanz, 2003] Pavón, J., and Gómez-Sanz, J. (2003). Agent Oriented Software Engineering with INGENIAS. *Multi-Agent Systems and Applications III*, pages 394–403, Springer-Verlag.

- [Payton et al., 2002] Payton, D., Estkowski, E., and Howard, R. (2002). Progress in Pheromone Robotics. *Proceedings of the 7th International Conference on Intelligent Autonomous Systems*.
- [Pister, 2001] Pister, K. (2001). Smart Dust: Autonomous Sensing and Communication in a Cubic Millimeter. Available at <http://robotics.eecs.berkeley.edu/~pister/SmartDust>.
- [Plosch and Pichler, 1999] Plosch, R., and Pichler, J. (1999). Contracts: From Analysis to C++ Implementation. *IEEE Computer*, pages 248–257.
- [Poor, 2001] Poor, R. D. (2001). Embedded Networks: Pervasive, Low-Power, Wireless Connectivity. Ph.D. Thesis, MIT.
- [Popovici et al., 2003] Popovici, A., Frei, A., and Alonso, G. (2003). A Proactive Middleware Platform for Mobile Computing. *Proceedings of the 4th International Middleware Conference (Middleware'2003)*, Rio de Janeiro, Brazil.
- [Poslad et al., 2000] Poslad, S., Buckle, P., and Hadingham, R. (2000). The FIPA-OS Agent Platform: Open Source for Open Standards. Available at <http://fipa-os.sourceforge.net>.
- [Poutakidis et al., 2002] Poutakidis, D., Padgham, L., and Winikoff, M. (2002). Debugging Multi-Agent Systems using Design Artifacts: The Case of Interaction Protocols. *Proceedings of the 1st International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 960–967, ACM Press.
- [Poutakidis et al., 2003] Poutakidis, D., Padgham, L., and Winikoff, M. (2003). An Exploration of Bugs and Debugging in Multi-Agent Systems. *Proceedings of the 14th International Symposium on Methodologies for Intelligent Systems (ISMIS)*, Maebashi City, Japan.
- [Pratt et al., 1999] Pratt, D. R., Ragusa L. C., and von der Lippe, S. (1999). Composability as an Architecture Driver. *Proceedings of the International Conference on Interservice/Industry Training, Simulation and Education*, Orlando, Florida.
- [Pree, 1995] Pree, W. (1995). State-of-the-Art Design Pattern Approaches: An Overview. *Proceedings of Technology of Object-Oriented Languages and Systems (TOOLS '95)*.
- [Pressman, 1982] Pressman, R. S. (1982). *Software Engineering: A Practitioner's Approach*. McGraw-Hill.

- [Priyantha et al., 2000] Priyantha, N. B., Chakraborty, A., and Balakrishnan, H. (2000). The Cricket Location-Support System. *Proceedings of MobiCom 2000*.
- [Purvis et al., 2002] Purvis, M., Cranefield, S., Nowostawski, M., Ward, R., Carter, D., and Oliveira, M.A., (2002). Agentcities Interaction Using the Opal Platform. *Proceedings of Agentcities: Research in Large-Scale Open Agents Environments*.
- [Ralph and Shephard, 2001] Ralph, D., and Shephard, C. G. (2001). Services via Mobility Portals. *BT Technology Journal*, 19(1).
- [Rana and Moreau, 2000] Rana, O. F., and Moreau, L. (2000). Issues in Building Agent based Computational Grids. *Third Workshop of the UK Special Interest Group on Multi-Agent Systems (UKMAS'2000)*, Oxford, UK.
- [Rana and Walker, 2000] Rana, O. F., and Walker, D. W. (2000). The Agent Grid': Agent-Based Resource Integration in PSEs. *Proceedings of the 16th IMACS World Congress on Scientific Computation, Applied Mathematics and Simulation*, Lausanne, Switzerland.
- [Rao, 1996] Rao, A. S. (1996). AgentSpeak(L): BDI Agents Speak out in a Logical Computable Language, van der Velde, W, and Perram, J. (Eds.) *Proceedings of the 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW-96)*, pages 42–55, Springer-Verlag.
- [Rao and Georgeff, 1991] Rao, A. S., and Georgeff, M. P. (1991). Modeling Rational Agents within a BDI-Architecture. Allen, J., Fikes, R., and Sandewall, E. (Eds.) *Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning (KR'91)*, pages 473–484, Morgan Kaufmann.
- [Ratsimor et al., 2002] Ratsimor, O., Chakraborty, D., Tolia, S., Kushraj, D., Kunjithapatham, A., Gupta, G., Joshi, A., and Finin, T. (2002). Allia: Alliance-based Service Discovery for Ad-Hoc Environments. *Proceedings of the 2nd ACM Workshop on Mobile Commerce*, Atlanta, USA.
- [Rich and Knight, 1990] Rich, E., and Knight, K. (1990). *Artificial Intelligence*. McGraw-Hill.
- [Ricci et al., 2002] Ricci, A., Omicini, A., and Denti, E. (2002). Virtual Enterprises and Workflow Management as Agent Coordination Issues. *International Journal of Cooperative Information Systems*, 11 (3/4), pages 355–379.
- [Ricci et al., 2003] Ricci, A., Omicini, A., and Denti, E. (2003). Activity Theory as a Framework for MAS Coordination. Petta, P., Tolksdorf, R., and

- Zambonelli, F. (Eds.) *Engineering Societies in the Agents World III*, pages 96–110. Springer-Verlag.
- [Ricordel and Demazeau, 2000] Ricordel, P.-M., and Demazeau, Y. (2000). From Analysis to Deployment: A Multi-agent Platform Survey. *Working Notes of the First International Workshop on Engineering Societies in the Agents' World (ESAW-00)*, pages 93–105.
- [Rodríguez et al., 1998] Rodríguez, J. A., Martín, F. J., Noriega, P., García, P., and Sierra, C. (1998). Towards a Test-Bed for Trading Agents in Electronic Auction Markets. *AI Communications*, 11(1), pages 5–19.
- [Roli, 2002] Roli, F., and Zambonelli, A. (2002). Emergent Behaviors in Dissipative Cellular Automata. *Proceedings of the 5th International Conference on Cellular Automata for Research and Industry (ACRI 2002)*, Geneva, Italy.
- [Rosenschein and Kaelbling, 1995] Rosenschein, S. J., and Kaelbling, L. P. (1995). A Situated View of Representation and Control. *Artificial Intelligence*, 73(1–2), pages 149–173.
- [Royal Society of London, 2003] Royal Society of London (2003). Self-Organization: The Quest for the Origin and Evolution of Structure. *Philosophical Transactions of the Royal Society of London*.
- [Russell and Norvig, 1995] Russell, S., and Norvig, P. (1995). *Artificial Intelligence: A Modern Approach*. Prentice-Hall International.
- [Ruth and Hannon, 1997] Ruth, M., and Hannon, B. (1997). *Modelling Dynamic Economic Systems*. Springer-Verlag.
- [Sadri and Toni, 1999] Sadri, F., and Toni, F. (1999). Computational Logic and Multiagent Systems: A Roadmap. Technical report, DFKI.
- [Saeki, 1994] Saeki, M. (1994). Software Specification & Design Methods and Method Engineering. *International Journal of Software Engineering and Knowledge Engineering*.
- [Saha and Mukherjee, 2003] Saha, D., and Mukherjee, A. (2003). Pervasive Computing: A Paradigm for the 21st Century. *IEEE Computer*, 36(3).
- [Sandhu et al., 1996] Sandhu, R. S., Coyne, E. J., Feinstein, H. L., and Youman, C. E. (1996). Role-Based Access Control Models. *IEEE Computer*, 29(2), pages 38–47.
- [Sashima et al., 2002] Sashima, A., Kurumatani, K., and Izumi, N. (2002). Physically-Grounding Agents in Ubiquitous Computing. *Proceedings of Joint Agent Workshop (JAWS'2002)*, Connecticut, USA.

- [Satyanarayanan, 2001] Satyanarayanan, M. (2001). Pervasive Computing: Vision and Challenges. *IEEE Personal Communications*, 8(4).
- [Sauter et al., 2002] Sauter, J. A., Matthews, R., Parunak, H. V. D., and Brueckner, S. (2002). Evolving Adaptive Pheromone Path Planning Mechanisms. *Proceedings of Autonomous Agents and Multiagent Systems*, pages 434–440.
- [Schmidt and Beigl, 1998] Schmidt, A., and Beigl, M. (1998). New Challenges of Ubiquitous Computing and Augmented Reality. *Proceedings of the 5th CaberNet Radicals Workshop*, Porto, Portugal.
- [Schmidt and Simone, 1996] Schmidt, K., and Simone, C. (1996). Coordination Mechanisms: Towards a Conceptual Foundation of CSCW Systems Design. *Computer Supported Cooperative Work (CSCW)*, 5(2–3), pages 155–200.
- [Schmidt and Simone, 2000] Schmidt, K., and Simone, C. (2000). Mind the Gap! Towards a Unified View of CSCW. Dieng, R., Giboin, A., Karsenty, L., and de Michelis, G. (Eds.) *Designing Cooperative Systems: The Use of Theories and Models*, pages 205–221.
- [Schumacher, 2001] Schumacher, M. (2001). *Objective Coordination in Multi-Agent System Engineering – Design and Implementation*, Springer-Verlag.
- [Schweitzer and Zimmermann, 2001] Schweitzer, F., and Zimmermann, J. (2001). Communication and Self-Organization in Complex Systems: A Basic Approach. Fischer, M. M., and Fröhlich, J. (Eds.) *Knowledge, Complexity and Innovation Systems*, pages 275–296. Springer-Verlag.
- [Scott, 1998] Scott, W. R. (1998). *Organizations: Rational, Natural, and Open Systems*. Prentice-Hall International.
- [Searle, 1969] Searle, J. (1969). *Speech Acts: An Essay in the Philosophy of Language*. Cambridge University Press.
- [Self and DeLoach, 2003] Self, A., and DeLoach, S. A. (2003). Designing and Specifying Mobility within the Multiagent Systems Engineering Methodology. *Proceedings of the 8th ACM Symposium on Applied Computing*.
- [Sen and Weiss, 1999] Sen, S., and Weiss, G. (1999). Learning in Multiagent Systems. Weiss, G. (Ed.) *Multiagent Systems*, pages 259–299, The MIT Press.

- [Servat and Drogoul, 2002] Servat, D., and Drogoul, A. (2002). Combining Amorphous Computing and Reactive Agent-Based Systems: A Paradigm for Pervasive Intelligence? *Proceedings of AAMAS' 02*, ACM Press.
- [Shaikhali et al., 2003] Shaikhali, A., Rana, O. F, Al-Ali, R., and Walker, D. W. (2003). UDDIe: An Extended Registry for Web Services. *Proceedings of the Workshop on Service Oriented Computing: Models, Architectures and Applications*. IEEE Computer Society Press.
- [Shalizi, 2001] Shalizi, C. R. (2001). *Causal Architecture, Complexity and Self-Organization in Time Series and Cellular Automata*. Ph.D. Thesis, University of Wisconsin.
- [Shaw and Garlan, 1996] Shaw, M., and Garlan, D. (1996). *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall International.
- [Shehory and Sturm, 2001] Shehory, O., and Sturm, A. (2001). Evaluation of Modeling Techniques for Agent-Based Systems. *Proceedings of the 5th International Conference on Autonomous Agents*, pages 624–631. ACM Press.
- [Shen and Nome, 1999] Shen, W., and Norrie, D. (1999). Agent-Based Systems for Intelligent Manufacturing: A State of the Art Survey. *International Journal on Knowledge and Information Systems*, 1(2), pages 129–156.
- [Shen et al., 2002] Shen, W. M., Salemi, M., and Will, P. (2002). Hormone-Inspired Adaptive Communication and Distributed Control for CONRO Self-Reconfigurable Robots. *IEEE Transactions on Robotics and Automation* 18(5), pages 1-12.
- [Shoham, 1991] Shoham, Y. (1991). Agent0: An Agent-Oriented Programming Language and its Interpreter. *Proceedings of AAAI-91*, pages 704–709.
- [Shoham, 1993] Shoham, Y. (1993). Agent-Oriented Programming. *Artificial Intelligence*, 60(1), pages 51–92.
- [Shoham and Tennenholtz, 1995] Shoham, Y., and Tennenholtz, M. (1995). On Social Laws for Artificial Agent Societies: Off-Line Design. *Artificial Intelligence*, 73(1–2), pages 231–252.
- [Sichman, 1998] Sichman, J. S. (1998). Depint: Dependence-Based Coalition Formation in an Open Multi-Agent Scenario. *Journal of Artificial Societies and Social Simulation*, 1(2).

- [Sichman and Demazeau, 2001] Sichman, J. S., and Demazeau, Y. (2001). On Social Reasoning in Multi-Agent Systems. *Revista Iberoamericana de Inteligencia Artificial*, 3(13), pages 68–84.
- [Sichman et al., 1994] Sichman, J. S., Conte, R., Demazeau, Y., and Castelfranchi, C. (1994). A Social Reasoning Mechanism based on Dependence Networks. *Proceedings of the European Conference on Cognitive Science*.
- [Sierra et al., 1997] Sierra, C., Faratin, P., and Jennings, N. R. (1997). A Service-Oriented Negotiation Model between Autonomous Agents. *Proceedings of MAAMAW'97*, pages 17–35.
- [Siewiorek, 2002] Siewiorek, D. P. (2002). New Frontiers of Application Design. *Communications of the ACM*, 45(12).
- [Singh, 1997] Singh, M. P. (1997). Formal Methods in DAI: Logic based Representation and Reasoning. *Multiagent Systems – A Modern Approach to Distributed Artificial Intelligence*, pages 331–376.
- [Singh, 1998] Singh, M. P. (1998). Agent Communication Languages: Rethinking the Principles. *IEEE Computer*, 31(12), pages 40–47.
- [Singh, 1999a] Singh, M. P. (1999a). An ontology for Commitments in Multiagent Systems: Toward a Unification of Normative Concepts. *Artificial Intelligence and Law*, 7, pages 97–113.
- [Singh, 1999b] Singh, M. P. (1999b). Write Asynchronous, Run Synchronous. *IEEE Internet Computing*, 3(2), pages 4–5.
- [Singh, 2000] Singh, M. P. (2000). Synthesizing Coordination Requirements for Heterogeneous Autonomous Agents. *International Journal of Autonomous Agents and Multi-Agent Systems*, 3(2), pages 107–132.
- [Singh, 2002] Singh, M. P. (2002). The Pragmatic Web. *IEEE Internet Computing*, 6(3), pages 4–5.
- [Singh, 2003] Singh, M. P. (2003). Distributed Enactment of Multiagent Workflows: Temporal Logic for Service Composition. *Proceedings of the 2nd International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 907–914. ACM Press.
- [Sommerville, 2001] Sommerville, I. (2001). *Software Engineering*. Addison-Wesley.
- [Sousa and Garlan, 2002] Sousa, J. P., and Garlan, D. (2002). Aura: An Architectural Framework for User Mobility in Ubiquitous Computing Environments. *Proceedings of the 3rd Working IEEE/IFIP Conference on Software Architecture (WICSA'2002)*, Montreal, CA.

- [Sparkman et al., 2001] Sparkman, C. H., DeLoach, S. A., and Self, A. L. (2001). Automated Derivation of Complex Agent Architectures from Analysis Specifications. *Proceedings of the 2nd International Workshop on Agent-Oriented Software Engineering*, Montreal, CA.
- [Spivey, 1992] Spivey, J. M. (1992). *The Z Notation*. Prentice-Hall International.
- [Sreenath and Singh, 2003] Sreenath, R. M., and Singh, M. P. (2003). Agent-Based Service Selection. *Journal on Web Semantics*, 1(1).
- [Stafford, 2003] Stafford, T. S. (2003). E-Services. *Communications of the ACM*, 46(6), pages 26–34. ACM Press.
- [Stanford, 2003] Stanford, V. (2003). Pervasive Computing Puts Food on the Table. *IEEE Pervasive Computing Magazine*, 2(1).
- [Stohr and Zhao, 2001] Stohr, E. A., and Zhao, J. L. (2001). Workflow Automation: Overview and Research Issues. *Information Systems Frontiers*, 3(3), pages 281–296.
- [Stone and Veloso, 2000] Stone, P., and Veloso, M. (2000). Multiagent Systems: A Survey from a Machine Learning Perspective. *Autonomous Robots*, 8(3), pages 345–383.
- [Sturm and Shehory, 2003] Sturm, A., and Shehory, O. (2003). A Framework for Evaluating Agent-Oriented Methodologies. Giorgini, P., and Winikoff, M. (Eds.) *Proceedings of the 5th International Bi-Conference Workshop on Agent-Oriented Information Systems*, pages 60–67.
- [Sturm et al., 2003] Sturm, A., Dori, D., and Shehory, O. (2003). Single-Model Method for Specifying Multi-Agent Systems. *Proceeding of 2nd International Joint Conference on Autonomous Agents and Multi Agent Systems*.
- [Suhail, 1998] Suhail, A. (1998). *CORBA Programming Unleashed*. Sams.
- [Sycara and Klusch, 2001] Sycara, K., and Klusch, M. (2001). Brokering and Matchmaking for Coordination of Agent Societies: A Survey. *Coordination of Internet Agents*, Springer-Verlag.
- [Sycara et al., 1999] Sycara, K., Klusch, M., Idof, S., and Lu, J (1999). Dynamic Service Matchmaking among Agents in Open Information Environments. *Journal ACM SIGMOD Record*.
- [Sycara et al., 2002] Sycara, K., Widoff, S., Klusch, M., Lu, J. (2002). LARKS: Dynamic Matchmaking Among Heterogeneous Software Agents

- in Cyberspace. *International Journal of Autonomous Agents and Multi-Agent Systems*, 5, pages 173–203.
- [Sycara et al., 2003] Sycara, K., Paolucci, M., van Velsen, M., and Giampapa, J. (2003). The RETSINA MAS Infrastructure. *International Journal on Autonomous Agents and Multi-Agent Systems*, 7(1–2), pages 29–48.
- [Szyperski, 1998] Szyperski, C. (1998). *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley.
- [Tambe et al., 2000] Tambe, M., Pynadath, D. V., Chauvat, N., Das, A., and Kaminka, G. A. (2000). Adaptive Agent Architectures for Heterogeneous Team Members. *Proceedings of the 4th International Conference on Multi-Agent Systems (ICMAS 2000)*, pages 301–308, Boston, USA.
- [Tennenhouse, 2000] Tennenhouse, D. (2000). Embedding the Internet: Proactive Computing. *Communications of the ACM*, 43(5), pages 36–42.
- [Theune et al., 2003] Theune, M., Faas, S., Heylen D., and Nijholt, A. (2003). The Virtual Storyteller: Story Creation by Intelligent Agents. *Proceedings of Technologies for Interactive Digital Storytelling and Entertainment*, pages 204–215.
- [Thomas, 1993] Thomas, S. R. (1993). *PLACA: An Agent Oriented Programming Language*. Ph.D. Thesis, Stanford University.
- [Thomas, 1995] Thomas, S. R. (1995). The PLACA Agent Programming Language. Wooldridge, M. J., and Jennings, N. R. (Eds.) *Intelligent Agents*, pages 355–370. Springer-Verlag.
- [Tolvanen, 1998] Tolvanen, J.-P. (1998). *Incremental Method Engineering with Modeling Tools: Theoretical Principles and Empirical Evidence*. Ph.D. Thesis, University of Jyväskylä.
- [Trastour et al., 2001] Trastour, D., Bartolini, C., and Gonzalez-Castillo, J. (2001). A Semantic Web Approach to Service Description for Matchmaking of Services. *Proceedings of the International Semantic Web Working Symposium (SWWS)*.
- [van de Vijver, 1997] van de Vijver, G. (1997). Emergence et Explication. *Intellectica: Emergence and Explanation*, 2-25.
- [van der Hoek, 2001] van der Hoek, W. (2001). Logical Foundations of Agent-Based Computing. Luck, M., Marík, V., Stepánková, O., and Trappl, R. (Eds.) *Multi-Agent Systems and Applications*, Springer-Verlag.

- [van Harmelen et al., 2002] van Harmelen, F., Horrocks, I., Clark, P., Patel-Schneider, P. F., Uschold, M., Rousset, M.-C., Hendler, J., and Schreiber, G. (2002). Ontologies' KISSES in Standardization. *IEEE Intelligent Systems*, 17, pages 70–79.
- [van Lamsweerde, 2000] van Lamsweerde, A. (2000). Requirements Engineering in the Year 00: A Research Perspective. *Proceedings of the 22nd International Conference on Software Engineering (ICSE'00)*, pages 5–19.
- [van Lamsweerde, 2001] van Lamsweerde, A. (2001). Goal-Oriented Requirements Engineering: A Guided Tour. *Proceedings of the 5th IEEE International Symposium on Requirements Engineering (RE'01)*, pages 249–263.
- [van Lamsweerde and Massonet, 1995] van Lamsweerde, R., Darimont, A., and Massonet, P. (1995). Goal-Directed Elaboration of Requirements for a Meeting Scheduler: Problems and Lessons Learnt. *Proceedings of the 5th IEEE International Symposium on Requirements Engineering (RE'01)*, pages 194–203.
- [van Roy and Haridi, 1999] van Roy, P., and Haridi, S. (1999). Mozart: A Programming System for Agent Applications. *Proceedings of the International Workshop on Distributed and Internet Programming with Logic and Constraint Languages*.
- [Varga et al., 1994] Varga, L. Z., Jennings, N. R., and Cockburn, D. (1994). Integrating Intelligent Systems into a Cooperating Community for Electricity Distribution Management. *International Journal of Expert Systems with Applications*, 7(4), pages 563–579.
- [Venkatraman and Singh, 1999] Venkatraman, M., and Singh, M. P. (1999). Verifying Compliance with Commitment Protocols: Enabling Open Web-Based Multiagent Systems. *International Journal on Autonomous Agents and Multi-Agent Systems*, 2(3), pages 217–236.
- [Viroli and Omicini, 2003] Viroli, M., and Omicini, A. (2003). Coordination as a Service: Ontological and Formal Foundation. Brogi, A., and Jacquet, J.-M. (Eds.) *Foundations of Coordination Languages and Software Architecture*, Elsevier Science.
- [Visser et al., 2000] Visser, W., Park, S., and Penix, J. (2000). Using Predicate Abstraction to Reduce Object-Oriented Programs for Model Checking. *Proceedings of the 3rd Workshop on Formal Methods in Software Practice*, pages 3–182. ACM Press.

- [von Bertalanffy, 1968] von Bertalanffy, L. (1968). *General System Theory*. George Braziller.
- [von Martial, 1992] von Martial, F. (1992). *Co-ordinating Plans of Autonomous Agents*, Springer-Verlag.
- [Vouk and Singh, 1997] Vouk, M. A., and Singh, M. P. (1997). Quality of Service and Scientific Workflows. Boisvert, R. F. (Ed.) *Quality of Numerical Software: Assessment and Enhancements*, pages 77–89.
- [Vygotskij, 1978] Vygotskij, L. S. (1978). *Mind and Society*. Harvard University Press.
- [Walshe et al., 2000] Walshe, D., Kennedy, J., Corley, S., Koudouridis, G., Laenen, F. V., Ouzounis, V., Garijo, F., and Gomez-Sanz, J. (2000). Eurcom P815: An Interoperable Architecture for Agent-Oriented Management. Arabnia, H. R. (Ed.) *Proceedings of IC-AI'2000*, CSREA Press.
- [Wan and Singh, 2003] Wan, F., and Singh, M. P. (2003). Commitments and Causality for Multiagent Design. *Proceedings of the 2nd International Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*, pages 749–756, ACM Press.
- [Wagner, 2002] Wagner, G. (2002). A UML Profile for External AOR Models. *Proceedings of the 3rd International Workshop on Agent-Oriented Software Engineering*.
- [Wegner, 1997] Wegner, P. (1997). Why Interaction is More Powerful than Computing. *Communications of the ACM*, 40(5), pages 80–91.
- [Weiser, 1991] Weiser, M. (1991). The Computer for the Twenty-First Century. *Scientific American*.
- [Weiser, 1993] Weiser, M. (1993). Some Computer Science Problems in Ubiquitous Computing. *Communications of the ACM*, 36(7).
- [Weiser, 1996] Weiser, M., and Brown, J. S. (1996). Designing Calm Technology. *PowerGrid Journal*, 1-1.
- [Weiss, 1999] Weiss, G. (Ed.) (1999). *Multiagent Systems. A Modern Approach to Distributed Artificial Intelligence*. The MIT Press.
- [Weiss, 2001] Weiss, R. (2001). *Cellular Computation and Communications Using Engineered Genetic Regulatory Networks*. Ph.D. Thesis, MIT.
- [Weiss, 2003] Weiss, G. (2003). Agent Orientation in Software Engineering. *Knowledge Engineering Review*, 16(4), pages 349–373.

- [Weiss et al., 2003] Weiß, G., Rovatsos, M., and Nickles, M. (2003). Capturing Agent Autonomy in Roles and XML. *Proceedings of the 2nd International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 105–112, ACM Press.
- [Whitaker, 2003] Whitaker, R. (2003). ACM SIGGROUP: Self-Organization, Autopoiesis, and Enterprises. Available at <http://www.acm.org/sigois/auto/Main.html>.
- [Wieland, 2003] Wieland, K. (2003). The Long Road to 3G. *International Telecommunications Magazine*, 37(2).
- [Winikoff et al., 2001] Winikoff, M., Padgham, L., and Harland, J. (2001). Simplifying the Development of Intelligent Agents. *Proceedings of AI2001*, pages 555–568. Springer-Verlag.
- [Wolpert, 1998] Wolpert, D. H. (1998). *Principles of Development*. Oxford University Press, UK.
- [Wolpert and MacReady 1995] Wolpert, D. H., and MacReady, W. G. (1995). No Free Lunch Theorems for Search. Technical Report SFI-TR-95-02-010, Santa Fe Institute.
- [Wood and DeLoach, 2001] Wood, M., and DeLoach, S. A. (2001). An Overview of the Multiagent Systems Engineering Methodology. Ciancarini, P., and Wooldridge, M. J. (Eds.) *Agent-oriented software engineering*, pages 207–222, Springer-Verlag,
- [Wooldridge, 1992] Wooldridge, M. J. (1992). *The Logical Modelling of Computational Multi-Agent Systems*. Ph.D. Thesis, UMIST.
- [Wooldridge, 1997] Wooldridge, M. J. (1997). Agent-Based Software Engineering. *IEE Proceedings Software Engineering*, 144(1), pages 26–37.
- [Wooldridge, 2000] Wooldridge, M. J. (2000). Intelligent Agents. Weiss, G. (Ed) *Multiagent Systems*, pages 27–78, The MIT Press.
- [Wooldridge, 2002] Wooldridge, M. J. (2002). *An Introduction to Multi-Agent Systems*. John Wiley & Sons.
- [Wooldridge and Ciancarini, 2000] Wooldridge, M. J., and Ciancarini, P. (2000). Agent-Oriented Software Engineering: The State of the Art. Ciancarini, P., and Wooldridge, M. J. (Eds.) *Proceedings of the 1st Workshop on Agent-Oriented Software Engineering*, pages 1–28, Springer-Verlag.
- [Wooldridge and Jennings, 1995a] Wooldridge, M. J., and Jennings, N. R. (1995a). Intelligent Agents: Theory and Practice. *Knowledge Engineering Review*, 10(2), pages 115–152.

- [Wooldridge and Jennings, 1995b] Wooldridge, M. J., and Jennings, N. R. (1995b). Agent Theories, Architectures, and Languages: A Survey. Wooldridge, M. J., and Jennings, N. R. (Eds.) *Intelligent Agents*, pages 1–39, Springer-Verlag.
- [Wooldridge and Jennings, 1998] Wooldridge, M. J., and Jennings, N. R. (1998). Pitfalls of Agent-Oriented Development. Sycara, K., and Wooldridge, M. J. (Eds.) *Proceedings of the 2nd International Conference on Autonomous Agents*, ACM Press.
- [Wooldridge et al., 1999] Wooldridge, M. J., Jennings, N. R., and Kinny, D. (1999). A Methodology for Agent-Oriented Analysis and Design. Etzioni, O., Müller, J. P., and Bradshaw, J. M. (Eds.) *Proceedings of the 3rd International Conference on Autonomous Agents*, pages 69–76, ACM Press.
- [Wooldridge et al., 2002a] Wooldridge, M. J., Fisher, M., Huget, M.-P., and Parsons, S. (2002a). Model Checking Multi-Agent Systems with MABLE. *Proceedings of the 1st International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 952–959, ACM Press.
- [Wooldridge et al., 2000b] Wooldridge, M. J., Jennings, N. R., and Kinny, D. (2000b). The Gaia Methodology for Agent-Oriented Analysis and Design. *International Journal of Autonomous Agents and Multi Agent Systems*, 3(3), pages 285–312.
- [Wooldridge et al., 2002c] Wooldridge, M. J., Weis, G., and Ciancarini, P. (Eds.) (2002c). *Agent-Oriented Software Engineering II*, Springer-Verlag.
- [Xing and Singh, 2003] Xing, J., and Singh, M. P. (2003). Engineering Commitment-Based Multiagent Systems: A Temporal Logic Approach. *Proceedings of the 2nd International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 891–898, ACM Press.
- [Xu et al., 2002] Xu, D., Volz, R., loerger, T., and Yen, J. (2002). Modeling and Verifying Multi-agent Behaviors using Predicate/Transition Nets. *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering*, pages 193–200, ACM Press.
- [Yolum and Singh, 2002] Yolum, P., and Singh, M. P. (2002). Flexible Protocol Specification and Execution: Applying Event Calculus Planning using Commitments. *Proceedings of the 1st International Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*, pages 527–534, ACM Press.
- [Yu, 1995] Yu, E. (1995). *Modelling Strategic Relationships for Process Reengineering*. Ph.D. Thesis, University of Toronto.

- [Yu, 1997a] Yu, E. (1997a). Towards Modelling and Reasoning Support for Early-Phase Requirements Engineering. *Proceedings of 3rd International Symposium on Requirements Engineering*, pages 226–235.
- [Yu, 1997b] Yu, E. (1997b). Why Agent-Oriented Requirements Engineering? *Proceedings of 3rd International Workshop on Requirements Engineering*.
- [Yu, 1999] Yu, E. (1999). Strategic Modelling for Enterprise Integration. *Proceedings 14th World Congress of the International Federation of Automatic Control*.
- [Yu, 2001] Yu, E. (2001). Agent-Oriented Modeling: Software versus the World. *Agent-Oriented Software Engineering II*, Springer-Verlag.
- [Yu and Cysneiros, 2002] Yu, E., and Cysneiros, L. M. (2002). Agent-Oriented Methodologies – Towards a Challenge Exemplar. *Proceedings of the 4th International Bi-Conference Workshop on Agent-Oriented Information Systems*.
- [Yu and Liu, 2002] Yu, E., and Liu, L. (2002). Designing Web-Based Systems in Social Context: A Goal and Scenario based Approach. *Advanced Information Systems Engineering*, pages 37–51, Springer-Verlag.
- [Yu and Liu, 2003] Yu, E., and Liu, L. (2003). Organization Modelling Environment. Available at <http://www.cs.toronto.edu/km/ome>.
- [Yu and Mylopoulos, 1998] Yu, E., and Mylopoulos, J. (1998). Why Goal-Oriented Requirements Engineering? *Proceedings of the 4th International Workshop on Requirements Engineering*, pages 15–22.
- [Yu and Singh, 2002] Yu, B., and Singh, M. P. (2002). An Evidential Model of Distributed Reputation Management. *Proceedings of the 1st International Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*, pages 294–301. ACM Press.
- [Yu et al., 2001] Yu, E., Liu, L., and Li, Y. (2001). Modelling Strategic Actor Relationships to Support Intellectual Property Management. *Conceptual Modeling*, pages 164–178, Springer-Verlag.
- [Yunos et al., 2003] Yunos, H. M., Gao, J. Z., and Shim, S. (2003). Wireless Advertising's Challenges and Opportunities. *IEEE Computer*, 26(5).
- [Zambonelli and Parunak, 2002] Zambonelli, F., and Parunak, H. V. D. (2002). Sign of a Revolution in Computer Science and Software Engineering. *Proceedings of the 3rd International Workshop on Engineering Societies in the Agents' World*, Springer-Verlag.

- [Zambonelli and Parunak, 2003] Zambonelli, F., and Parunak, H. V. D. (2003). Toward a Change of Paradigm in Computer Science and Software Engineering: A Synthesis. *The Knowledge Engineering Review*, 18, 2003.
- [Zambonelli et al., 2000] Zambonelli, F., Jennings, N. R., and Wooldridge, M. J. (2000). Organisational Abstractions for the Analysis and Design of Multi-Agent Systems. Ciancarini, P., and Wooldridge, M. J. (Eds.) *Agent-Oriented Software Engineering*, pages 127–141, Springer-Verlag.
- [Zambonelli et al., 2001a] Zambonelli, F., Jennings, N. R., Omicini, A., and Wooldridge, M. J. (2001a). Agent-Oriented Software Engineering for Internet Applications. *Coordination of Internet Agents: Models, Technologies, and Applications*, pages 326–346, Springer-Verlag.
- [Zambonelli et al., 2001b] Zambonelli, F., Jennings, N. R., and Wooldridge, M. J. (2001). Organizational Rules as an Abstraction for the Analysis and Design of Multiagent Systems. *International Journal of Software Engineering and Knowledge Engineering*, 11(4), pages 303–328.
- [Zambonelli et al., 2003] Zambonelli, F., Jennings, N. R., and Wooldridge, M.J. (2003). Developing Multiagent systems: The Gaia methodology. *ACM Transactions on Software Engineering and Methodology*, 12(3), pages 417–470.
- [Zave, 1997] Zave, P. (1997). Classification of Research Efforts in Requirements Engineering. *ACM Computing Surveys*, 29(4), pages 315–321.

Index

3APL, 57

AAII, 174

ABLE framework, 54

ACL (Agent Communication Language), 22, 47, 260, 417

Activity Theory, 283, 284

ADELFE, 40, 53, 154, 158, 232, 338

agent, 9, 20, 21, 36, 48, 90, 130, 179, 191, 248, 260, 305, 396, 398, 407, 416, 433, 434

agent

agent architecture, 45, 445

agent class, 117

agent lifeline, 240

agent model, 35

agent-based model, 197

cooperative agent, 159, 164, 167

types of agents, 408

agent class diagram, 248, 254

agent platform, 53

agent technology, 431, 432, 434

agent-based software development, 13

Agent0, 56

AgentBuilder, 42

AgentSpeak (L), 56

agentTool, 38, 43, 60, 122, 172

AIP (Agent Interaction Protocol), 166

AIP (Agent Interaction Protocol), 85

Albert (language), 37

AMAS (Adaptive MultiAgent Systems), 157, 158, 327

amorphous computing, 300, 304, 361

analysis, 34, 35, 41, 70, 71, 76, 108, 163, 181

AOSE (Agent Oriented Software Engineering), 19, 394

AOSE (Agent-Oriented Software Engineering), 3

April language, 55

ASL, 262

AUML, 40, 84, 166, 172, 173, 178, 237

autocatalysis, 358, 360

Autonomic Computing, 7

autonomy, 10, 14, 49, 129, 267, 398

Bee-gent, 262

CLIPS language, 55

co-construction, 285

commitment, 13, 15

communication, 12, 15, 22, 47, 260, 417

complex system, 322, 433

component, 21

Concurrent-METATEM, 39, 55

ConGOLOG, 39, 56

conversation, 117, 119

cooperation, 284, 285, 328

coordination, 48, 273, 274, 286

coordination artifact, 284, 291

Data Coupling Diagram, 222

debugging, 58, 60, 229, 270, 439

decentralization, 351

delegation of responsibility, 23

dependency, 91, 274

deployment, 446

design, 34, 41, 71, 74, 77, 108, 117, 121, 165, 441

design

architectural design, 90, 93, 218, 222

architecture design, 423

detailed design, 90, 94, 218, 226

high-level design, 181

low-level design, 182

DESIRE, 44, 173

development environment, 42

Dia, 255

dialogic framework, 204

DISCERN, 386

distribution, 46, 351, 433, 441

dynamic, 11, 14, 352, 380

Electronic Institution Model, 197

emergence, 196, 308, 326, 347

emergence

emergent behaviour, 195

emergent functionality, 158

emergent phenomenon, 325

- environment, 12, 24, 75, 76, 87, 161, 162, 220, 279, 357, 382, 433
- Equation-Based Models, 196
- evaluation (of a methodology), 67, 129, 134, 138, 143
- evolutionary computation, 196, 210
- FIPA (Foudation for Intelligent Physical Agents), 260
- FIPA (Foundation for Intelligent Physical Agents), 20, 54, 239, 437
- FIPA-OS, 263
- Formal Tropos, 94
- functional adequacy, 164, 328, 334
- Gaia, 39, 52, 66, 69, 124, 134, 173, 178, 220, 231
- goal, 58, 76, 108, 110, 179, 220
- goal analysis, 102
- Grasshopper, 263
- grid, 394, 413, 414, 447
- Growing Point Language, 310, 314
- heterogeneity, 8, 11, 14, 441
- i*, 36, 58, 90
- implementation, 34, 55
- implementation tool, 444
- infrastructure, 273, 278
- INGENIAS, 41,43, 172, 194
- interaction, 24, 179, 180, 187, 275, 281, 382
- interaction
 - interaction diagram, 225
 - interaction model, 74, 77
 - thread of interaction, 241
- JACK, 229
- JADE, 44, 48, 54, 61,264
- KAOS, 37
- language, 55
- lifecycle, 66, 127, 132, 196
- Lisp, 56
- MABLE, 60
- MADKIT platform, 41, 52, 54, 61
- MAML, 57
- MAS-CommonKADS, 38, 178
- MaSE, 37, 66, 107, 125, 143, 172, 232
- MASIF, 54
- MASSIVE, 173
- mediation, 382
- MESSAGE, 41, 52, 154, 173, 177, 232
- message, 130, 243, 267
- meta-model, 41, 81
- methodology, 65, 127, 153, 435, 436
- Microsoft Visio, 255
- mobile computing, 399
- modelling language, 66, 127
- morphogen gradients, 306
- Mozart language, 56
- multiagent system, 107, 197, 378, 397, 435, 441
- NCS (Non Cooperative Situations), 159, 167, 170, 328, 330, 335
- norm, 130, 197, 204, 205
- notation, 85, 87, 131,224, 238
- observation, 382
- online engineering, 378, 383
- ontology, 47, 198
- Opal, 263
- open architecture, 5, 6
- open system, 8, 301, 379, 380, 382, 441
- open system
 - a model of open computational system, 381
- OpenTool, 172, 173, 255
- organization, 52, 69, 130, 179, 181, 192, 252, 292, 344
- Organization Theory, 98
- organization view, 180
- organizational rules, 76, 77
- organizational structures, 76, 78, 87
- PASSI, 40,43, 172, 173
- performative structure, 204
- pervasive computing, 315, 399
- pheromone, 310
- Prolog, 56
- Prometheus, 155, 173,217
- Prometheus Design Tool, 228
- protocol, 13, 15, 80, 130, 190, 197, 226, 245
- requirement, 35, 181, 220
- requirement
 - early requirement, 90, 91
 - final requirement, 161
 - late requirement, 90, 92
 - preliminary requirement, 161
- RETSINA framework, 54
- ROADMAP, 79
- role, 36, 71, 76, 108, 109, 113, 115, 130, 179, 198, 240, 249
- RUP (Rational Unified Process), 161, 178, 181
- SADDE, 53, 154, 196, 232
- scene, 197, 198, 204
- self-assembly, 311
- self-organization, 52, 158, 300, 325, 331, 344–346, 348
- semantic composability, 27
- semantic extensibility, 28
- semantic interoperability, 25

- sequence diagrams, 112, 239, 253
- service, 74, 421, 422
- service discovery, 419, 420
- service-oriented approach, 415
- SOLACE, 385
- specification, 218,220
- stereotype, 167
- swarming, 300, 341–343, 353

- task, 108, 114, 115, 130, 179, 182, 291
- testing, 34,57,229,440
- TOTA, 316
- Tropos, 58,66, 90, 125, 138, 173, 220, 232
- TuCSoN, 293

- ubiquitous computing, 394, 395, 400, 401
- UDDI (Universal Description Discovery and Integration), 14,404, 415, 420,422
- UML, 40, 84, 173, 177, 178, 237
- use cases, 112, 162,221

- validation, 57, 383, 384,440
- verification, 57, 58, 94,440
- VIRTUE, 385

- Web services, 6,420

- Z, 38
- Zeus, 37, 42,60, 263