

ALMA MATER STUDIORUM
UNIVERSITÀ DEGLI STUDI DI BOLOGNA

Facoltà di Ingegneria
Dipartimento di Elettronica Informatica e Sistemistica
Dottorato in Ingegneria Elettronica, Informatica e delle Telecomunicazioni

META-MODELS, ENVIRONMENT AND LAYERS:
AGENT-ORIENTED ENGINEERING OF COMPLEX
SYSTEMS

Disciplinary Sector: ING-INF/05
Cicle XX

Candidate:

Dott. Ing. AMBRA MOLESINI

Supervisors:

Chiar.mo Prof. Ing. ANTONIO NATALI
Ill.mo Prof. Ing. ENRICO DENTI
Ill.mo Prof. Ing. ANDREA OMICINI

Coordinator:

Chiar.mo Prof. Ing. PAOLO BASSI

FINAL EXAM YEAR 2008

ALMA MATER STUDIORUM
UNIVERSITÀ DEGLI STUDI DI BOLOGNA

Facoltà di Ingegneria
Dipartimento di Elettronica Informatica e Sistemistica
Dottorato in Ingegneria Elettronica, Informatica e delle Telecomunicazioni

March 2008

Supervisors:

Chiar.mo Prof. Ing. ANTONIO NATALI
Ill.mo Prof. Ing. ENRICO DENTI
Ill.mo Prof. Ing. ANDREA OMICINI

External Reviewers:

Prof. MARIE-PIERRE GLEIZES
Prof. MICHAEL LUCK

Coordinator:

Chiar.mo Prof. Ing. PAOLO BASSI

Acknowledgement

This thesis is the result of three years spent with a wonderful research group: thanks to Enrico and Andrea for their guiding visions and stimuli, to Mirko and Alessandro for their suggestions and infinite knowledge, and to Matteo and Elena for the discussions, ideas and interactions. They have been guides as well as friends helping me along all the stages of my Ph.D adventure. Thanks to Antonio, who instilled in me the passion for the world of software engineering. Thanks to Marie-Pierre and Michael for the great help that they gave me for improving the thesis. This work is dedicated to my parents and my boyfriend Alessandro.

Contents

Abstract	xi
1 Preface	1
1.1 The Contributions	2
1.2 The Structure of the Thesis	4
I Background	7
2 The MAS Approach	9
2.1 The Agent Paradigm	9
2.1.1 Complex Systems	10
2.1.2 The Agents	11
2.1.3 The Multi-Agent Systems Architecture	14
2.2 Societies	16
2.3 The Environment	18
2.3.1 Environment Abstractions	18
2.3.2 Topology Abstractions	19
3 From SE To AOSE	21
3.1 Software	21
3.2 Traditional Software Engineering	22
3.2.1 Software Development Processes	23
3.2.2 Methodologies	25
3.2.3 Tools	26
3.3 Agent Oriented Software Engineering	27
3.3.1 Paradigm Shift	29
3.3.2 AOSE Ingredients	30
3.4 Summing up	33
4 Agent Oriented Methodologies	35
4.1 Methodologies for Agent-Oriented Systems	35
4.2 The most known AO Methodologies	37
4.2.1 Gaia	37
4.2.2 ADELFE	39
4.2.3 Tropos	41
4.2.4 PASSI	43
4.2.5 MaSE	44
4.2.6 INGENIAS	46

4.2.7	MESSAGE	49
4.2.8	Prometheus	51
4.3	Methodologies Comparison	54
4.3.1	Lifecycle Criteria	54
4.3.2	Notation Criteria	56
4.3.3	Summing up	57
 II Meta-models		 59
5	Meta-models & Languages	61
5.1	Meta-Models	61
5.2	Meta-Modelling Languages	65
5.2.1	Meta-Modelling Languages for Abstractions	65
5.2.2	Meta-Modelling Languages for Processes	68
5.3	Meta-Modelling Languages for Infrastructures	73
5.4	Summing up	76
 6	 AOSE & Meta-models	 77
6.1	PASSI	77
6.1.1	PASSI: Concepts Meta-model	77
6.1.2	PASSI: Process Meta-model	79
6.2	ADELFE	82
6.2.1	ADELFE: Concepts Meta-model	82
6.2.2	ADELFE: Process Meta-model	84
6.3	Tropos	86
6.3.1	Tropos: Concepts Meta-model	86
6.3.2	Tropos: Process Meta-model	87
6.4	Gaia	91
6.5	Summing up	93
 7	 The Agents & Artifacts Meta-Model	 95
7.1	A&A Meta-Model	95
7.2	Artifacts	97
7.2.1	Features	99
7.2.2	Taxonomy of artifacts	104
7.3	Workspaces	105
7.4	Summing up	106

III	Environment	107
8	AOSE & Environment	109
8.1	Classification of AO Methodologies	110
8.2	Strong-Env Methodologies	110
8.2.1	ADELFE	111
8.2.2	OperA+Environment	111
8.2.3	Strong-env Methodologies at a Glance	112
8.3	Weak-Env Methodologies	112
8.3.1	Gaia	112
8.3.2	PASSI	113
8.3.3	MESSAGE	113
8.3.4	INGENIAS	114
8.3.5	Prometheus	114
8.3.6	ROADMAP	115
8.3.7	Weak-env Methodologies at a Glance	115
8.4	No-Env Methodologies	115
8.4.1	Tropos	115
8.4.2	MaSE	116
8.4.3	No-env Methodologies at a Glance	116
8.5	Summing up	117
9	Environment in AO Methodologies	119
9.1	Environment and Topology Abstractions	119
9.2	From No-Env to Weak-Env Methodologies	121
9.2.1	Requirement Specification	121
9.2.2	Analysis	122
9.2.3	Design	123
9.2.4	An Example: Tropos from No-Env to Weak-Env	123
9.3	From Weak-Env to Strong-Env Methodologies	125
9.3.1	Requirement Specification	125
9.3.2	Analysis	125
9.3.3	Design	125
9.3.4	An Example: Gaia from Weak-Env to Strong-Env	126
9.4	Related Work	127
9.5	Summing up	128
10	AOSE & Infrastructures	129
10.1	Definitions	130
10.2	Infrastructures for MAS	131
10.2.1	Enabling vs. Governing Infrastructures	132

10.3	SE and Infrastructures	134
10.3.1	Infrastructure Selection	135
10.3.2	A Sketch of the State of the Art	136
10.4	AOSE & Infrastructures	137
10.4.1	Coordination, Organisation and Security	139
10.5	AO Infrastructures	140
10.5.1	JADE	141
10.5.2	TuCSoN	145
10.5.3	CArtAgO	150
10.5.4	TOTA	153
10.6	Infrastructures: Summing up	157
 IV Representation Complexity		 159
 11 Complex Systems		 161
11.1	Software Systems and Complexity	162
11.1.1	Features of Complex Software Systems	163
11.2	Complex Systems and Hierarchies	165
11.3	Complex Systems and Self-organisation	166
11.4	Holonic Systems: Hierarchies and Self-organisation	169
11.5	Summing up	172
 12 Managing System Complexity		 173
12.1	Middle-out as the de-facto Practice	174
12.2	System Complexity in OO Methodologies and Notations	175
12.2.1	Detail Decomposition in OPM	176
12.3	System Complexity in AO Methodologies	178
12.3.1	AO Methodologies & Layering	178
12.4	Layering Mechanisms for AO Methodologies: A First Insight	180
12.4.1	Zoom & Artifacts	181
12.5	Summing up	183
 V SODA		 185
 13 SODA: The Early Version		 187
13.1	The Analysis Phase	188
13.1.1	The Role Model	189
13.1.2	The Resource Model	190
13.1.3	The Interaction Model	190
13.1.4	Analysis: the outcome	191

13.2	The Design Phase	191
13.2.1	The Agent Model	192
13.2.2	The Society Model	192
13.2.3	The Environment Model	193
13.2.4	Design: the outcome	194
13.3	Meta-models	194
13.3.1	Meta-model in UML	195
13.3.2	Meta-model in OPM	196
13.3.3	Discussion	197
13.4	Limitations	199
14	SODA: The New Version	201
14.1	Motivations	201
14.2	The New Meta-model	202
14.3	Layering	205
14.4	The Analysis Phase	206
14.4.1	Requirements Analysis	207
14.4.2	From Requirements Analysis to Analysis	208
14.4.3	Analysis	209
14.5	The Design Phase	210
14.5.1	From Analysis To Architectural Design	210
14.5.2	Architectural Design	210
14.5.3	From Architectural Design to Detailed Design	212
14.5.4	Detailed Design	214
14.6	Summing up	215
15	The SODA Process	217
15.1	The process	218
15.2	The Analysis Phase	220
15.2.1	The Analysis Discipline	220
15.2.2	The Requirement Analysis step	224
15.2.3	The Analysis step	226
15.2.4	The Analysis Model	228
15.3	The Design Phase	229
15.3.1	The Design Discipline	229
15.3.2	The Architectural Design step	233
15.3.3	The Detailed Design step	235
15.3.4	The Design Model	237
15.4	Summing up	238

16 SODA & Infrastructures	239
16.1 From SODA to TuCSoN	239
16.2 From SODA to CArTAgO	241
16.3 From SODA to TOTA	242
16.4 Discussion	243
16.5 Summing up	244
17 Case Study	245
17.1 Conference Management Systems	246
17.2 CMS in the literature	247
17.3 CMS & Agent-Oriented Approach	249
17.4 Conference Management in SODA	252
17.4.1 Requirements Analysis	252
17.4.2 From Requirements Analysis to Analysis	254
17.4.3 Analysis	255
17.4.4 From Analysis to Architectural Design	257
17.4.5 Architectural Design	258
17.4.6 From Architectural Design to Detailed Design	259
17.4.7 Detailed Design	260
17.5 From the design to a TuCSoN-based implementation	261
17.6 Discussion	263
VI Conclusion	269
18 Conclusion and Research Directions	271
18.1 Summary of the Contributions	271
18.2 Research Directions	273
VII Appendix	277
A The Complete Case Study	279
A.1 Requirements Analysis	279
A.2 From Requirements Analysis to Analysis	282
A.3 Analysis	284
A.4 From Analysis to Architectural Design	289
A.5 Architectural Design	291
A.6 From Architectural Design to Detailed Design	297
A.7 Detailed Design	299
Bibliography	302

Abstract

Traditional software engineering approaches and metaphors fall short when applied to areas of growing relevance such as electronic commerce, enterprise resource planning, and mobile computing: such areas, in fact, generally call for open architectures that may evolve dynamically over time so as to accommodate new components and meet new requirements. This is probably one of the main reasons that the *agent* metaphor and the agent-oriented paradigm are gaining momentum in these areas.

This thesis deals with the engineering of complex software systems in terms of the agent paradigm. This paradigm is based on the notions of agent and systems of interacting agents as fundamental abstractions for designing, developing and managing at runtime typically distributed software systems. However, today the engineer often works with technologies that do not support the abstractions used in the design of the systems. For this reason the research on methodologies becomes the basic point in the scientific activity. Currently most agent-oriented methodologies are supported by small teams of academic researchers, and as a result, most of them are in an early stage and still in the first context of mostly “academic” approaches for agent-oriented systems development. Moreover, such methodologies are not well documented and very often defined and presented only by focusing on specific aspects of the methodology. The role played by meta-models becomes fundamental for comparing and evaluating the methodologies. In fact a meta-model specifies the concepts, rules and relationships used to define methodologies. Although it is possible to describe a methodology without an explicit meta-model, formalising the underpinning ideas of the methodology in question is valuable when checking its consistency or planning extensions or modifications. A good meta-model must address all the different aspects of a methodology, i.e. the process to be followed, the work products to be generated and those responsible for making all this happen. In turn, specifying the work products that must be developed implies defining the basic modelling building blocks from which they are built.

As a building block, the agent abstraction alone is not enough to fully model all the aspects related to multi-agent systems in a natural way. In particular, different perspectives exist on the role that environment plays within agent systems: however, it is clear at least that all non-agent elements of a multi-agent system are typically considered to be part of the multi-agent system environment. The key role of environment as a first-class abstraction in the engineering of multi-agent system is today generally acknowledged in the multi-agent system community, so environment should be explicitly accounted for in the engineering of multi-agent system, working as a new design dimension for agent-oriented methodologies. At least two main ingredients shape the environment: environment abstractions – entities of the environment encapsulating some functions –, and topology abstractions — entities of environment that represent the (either logical or physical) spatial structure. In addition, the engineering of non-trivial multi-agent systems requires principles and mechanisms for supporting the management of the system

representation complexity. These principles lead to the adoption of a multi-layered description, which could be used by designers to provide different levels of abstraction over multi-agent systems.

The research in these fields has led to the formulation of a new version of the **SODA** methodology where environment abstractions and layering principles are exploited for engineering multi-agent systems.

Keywords: *Multi-agent Systems, Agent-oriented Software Engineering, Agent-oriented Methodologies, Environment, Infrastructures, Meta-models, Layering Principle, SODA*

1

Preface

In the three years of PhD I have had the opportunity to continue on the studies and work started in the last years of the Masters course at my university, reflecting a deep attractiveness for the world of software engineering, and of its impact on the development of quality software. This fascination was triggered by Enrico Denti, my professor of the course on programming languages and my future master thesis advisor. The attractiveness was amplified by the research work of Ian Sommerville in the context of traditional Software Engineering, and Michael Wooldridge and Nicholas Jennings on Agent Oriented Software Engineering. The engineering attitude in analysis and design of systems pushed me to search for a new and powerful paradigm for the engineering of complex systems like the multi-agent systems paradigm. Then I discovered the research work on agent-oriented software engineering developed at DEIS by my thesis advisor, Antonio Natali, with Andrea Omicini and Enrico Denti. That was a non-returning point. The interest on agents in general and on agent-oriented software engineering in particular was enhanced to cover all the software lifecycle: in particular agent-oriented methodologies became the core of my investigations, as a fundamental element in the engineering of software systems. These investigations started with my Masters thesis, “Analisi e Progetto di un sistema multi-agente per l’interazione avanzata docente /studente: agenti e servizi” (“Analysis and design of a multi-agent system for advanced teacher/student interaction: agents and services”) where I have used the **SODA** methodology for designing the agents. This work can be considered a sort of preface of this PhD thesis. The PhD period then focussed on the improvement of **SODA**. The research investigations and work grew in a great interactive research team, led by Andrea Omicini, with Enrico Denti, Alessandro Ricci and Mirko Viroli, overall covering the whole spectrum from the development of suitable meta-models for agent-oriented methodologies down to the role of both the environment – as a new design dimension – and infrastructures for multi-agent systems – as a deployment context of the systems. The interaction between these aspects, mixing top-down and bottom-up issues, has been fundamental for the development of the research work. In particular the studies about the environment have led to introduce the Agents&Artifacts meta-model in **SODA** in order to improve the analysis and design of the environment in the methodology. The studies of infrastructures have led to provide me some guidelines

for filling up the conceptual gap among methodologies and infrastructures. Finally the studies on the complexity of the system representation have led to the development of a layering principle for representing the system along different layers of abstractions. Those works have been captured and expressed in this thesis with the development of a new version of **SODA**.

1.1 The Contributions

The main contribution of this thesis is the development of the new version of **SODA**, an agent-oriented (AO) methodology. In order to achieve this result, the following contributions have been established:

- *Methodologies* — in general a methodology for software development (*i*) defines the abstractions to use to model software – the mindset of the methodology – and (*ii*) disciplines the software process—what to produce and when. My contribution in this thesis is a detailed presentation of the state of art of AO methodologies: a survey illustrates the best known AO methodologies and their peculiarities, another presents the methodologies meta-models, and another one presents how environment is managed by AO methodologies.
- *Meta-models* — a meta-model is a precise definition of the constructs and rules for creating semantic models. The importance of meta-model becomes clear when studying the completeness and the expressiveness of a methodology, and when comparing different methodologies. In this thesis meta-models are used as a tool for understanding the deep semantics of the AO methodologies. In particular two different types of meta-model are adopted: a meta-model for describing the methodologies abstractions and their relationships and a meta-model for modelling the software development process. One further my contribution in this research is the study of a meta-modelling technique for infrastructures, aimed at representing both the infrastructures' concepts and the dynamics. This is done in order to fill the gap between methodologies and infrastructure. In the last years, research on AO methodologies and multi-agent system (MAS) infrastructures has developed along two opposite paths: while AO methodologies have essentially undergone a top-down evolution pushed by contributions from heterogeneous fields like human sciences, MAS (multi-agent system) infrastructures have mostly followed a bottom-up path growing from existing and widespread (typically object-oriented) technologies. This dichotomy has produced a conceptual gap between the proposed AO methodologies and the agent infrastructures actually available, as well as a technical gap in the MAS engineering practice, where methodologies are often built *ad hoc* out of MAS infrastructures, languages and tools. By allowing structural representation of abstractions to be captured along with their mutual relations, meta-models make

it possible to map design-time abstractions from AO methodologies upon run-time abstractions from MAS technologies, thus promoting a more coherent and effective practice in MAS engineering.

- *Environment* — the key role of environment as a first-class abstraction in the engineering of MAS is generally acknowledged in the MAS community. However, the support for the notion of environment in today AO methodologies is still either absent, weak, or incomplete at best. Current practice in MAS considers the environment as an implicit part of the MAS that is often dealt with in ad hoc way. Indeed, the environment should be considered as an *explicit* part of MAS, to be modelled and designed as a *first-class abstraction*. Since a commonly shared viewpoint is lacking, it is seemingly useful first of all to analyse several AO methodologies studying the support they provide for the environment. The aim of my study is to understand how each of them models and designs the environment, and the abstractions it provides. As first contribution, in this thesis I classify AO methodologies along the dimension of environment support, and group them in three different categories: (*strong-env*) strong environment viewpoint—methodologies that support both modelling and design of MAS environment; (*weak-env*) weak environment viewpoint—methodologies that support only the modelling of MAS environment; (*no-env*) no environment viewpoint—methodologies that do not explicitly model MAS environment. Next, as second contribution in this field, I show how an explicit notion of MAS environment could be generally introduced in any AO methodology. Starting from no-env AO methodologies, I suggest how to transform them in weak-env methodologies, and subsequently in strong-env methodologies.
- *Complexity management* — A complex system is a system composed of interconnected parts that as a whole exhibit one or more properties (behaviour among the possible properties) not obvious from the properties of the individual parts. Ten years ago, Moxley wrote [134] that for large software systems, the specification of the software was the hardest part of the problem. Today the situation remains unchanged. The problem has been attacked repeatedly using a variety of methods, and the accepted thinking about the best way of doing it is still evolving. Some system development methodologies have adopted the decomposition principle by breaking the system into a number of models – *aspect decomposition* – each dealing with a different aspect of the system. Some others have adopted the principle by breaking the system into different levels of abstraction—*detailed decomposition*. One further my contribution in this research is the study of a mechanism for managing the complexity in AO methodologies. In particular taking inspiration from the detailed decomposition I have conceived the *layering principle* adopted by **SODA** in order to manage the complexity in the systems representations.

1.2 The Structure of the Thesis

The thesis has been designed and developed according five hierarchical levels (see Figure 1.1), which reflect the development of my work during the Ph.D: background, meta-models, environment, the complexity of the system representation and the **SODA** methodology.

<ul style="list-style-type: none"> - Multi-Agent Systems: agents, societies, environment - Agent-Oriented Software Engineering - Agent-Oriented Methodologies 	Background
<ul style="list-style-type: none"> - Meta-modelling - Meta-modelling Languages - Meta-models and Agent-Oriented Methodologies - A&A Meta-model 	Meta-models
<ul style="list-style-type: none"> - Environment & AOSE - Environment & Agent-Oriented Methodologies - Agent-Oriented Infrastructures 	Environment
<ul style="list-style-type: none"> - Complex Systems - Hierarchies and Self-Organisation - Methodologies & Hierarchies 	Representation Complexity
<ul style="list-style-type: none"> - SODA & Meta-models - SODA & Layering - SODA & Artifacts - SODA & Infrastructures 	SODA




Figure 1.1: Structure of the Thesis

- *Part I: Background* – This part discusses in detail the background context of the thesis, that is the engineering of complex software systems through multi-agent systems and agent societies. First the main features and benefits of the paradigm are illustrated (Chapter 2), then the ingredients of software engineering in general and of agent-oriented software engineering in particular are presented (Chapter 3), finally a survey of the main agent-oriented methodologies is reported (Chapter 4);
- *Part II: Meta-Models* – This part discusses in detail the issues related to the use and the construction of the methodology meta-models. In particular in Chapter 5 the meta-models and the languages that could be used for expressing meta-models

are illustrated, while in Chapter 6 the meta-models of some main agent-oriented methodologies are presented, then in Chapter 7 the Agents & Artifacts (A&A) meta-model is presented as a support for the engineering of multi-agent systems;

- *Part III: Environment* — This part discusses in detail the issue related to the role of environment in the engineering of MASs. In particular in Chapter 8 the agent-oriented methodologies are classified according to the support they provide for the modelling and design of environment, while in Chapter 9 a method for introducing the modelling and the design of the environment in those agent-oriented methodologies that do not support or support in a partial way the environment engineering is presented. Then in Chapter 10 the agent-oriented infrastructures as a deployment context for the MAS are presented and the meta-models of some different agent-oriented infrastructures are reported;
- *Part IV: Representation Complexity* — This part discusses in detail the issues related to managing system complexity. In particular in Chapter 11 some theories related to the complex systems are presented and a way for representing MAS as composed of different layers of abstraction is illustrated. Then in Chapter 12 the approaches for managing system complexity adopted both in the object-oriented methodologies and in the agent-oriented methodologies are presented. In addition a simple layering mechanism for the agent-oriented methodologies is presented;
- *Part V: SODA* — This part discusses in detail the **SODA** methodology. In particular in Chapter 13 the early version of the methodologies with its relatives meta-models and its limitations is presented. In Chapter 14 the new reformulated version of **SODA** is presented that addresses the limitations presented in the previous version, supporting both the environment design and the managing of system complexity. Then Chapter 15 presents the **SODA** process modelled by SPEM (Software Process Engineering Meta-model) and Chapter 16 presents a method based on the meta-models mapping for filling the gap between methodologies and infrastructures. Finally in Chapter 17 a case study in order to illustrate the use of the methodology is reported;

The thesis concludes by drawing some conclusions and identifying some research directions promoted by the work (Chapter 18). In the Appendix A the complete case study introduced in Chapter 17 is reported.

Part I

Background

2

The MAS Approach

This chapter familiarises the readers with the concepts used throughout the thesis. In particular Section 2.1 introduces the concepts of agent (Subsection 2.1.2), complex system (Subsection 2.1.1) an multi-agent system (Subsection 2.1.3). Then Section 2.2 presents the concepts of society of agents while Section 2.3 introduces the environment for MASs and the concepts of environment abstraction (Subsection 2.3.1) and topology abstraction (Subsection 2.3.2).

2.1 The Agent Paradigm

Agents and multi-agent systems (MASs) are a powerful technology to face the complexity of a variety of today's ICT scenarios. For instance, several industrial experiences already testify to the advantages of using agents in manufacturing processes [17], Web services and Web-based computational markets [105], and distributed network management [55]. In addition, several studies advise on the possibility of exploiting agents and MASs as enabling technologies for a variety of future scenarios, i.e., pervasive computing, Grid computing [65], Semantic Web [109]. However, the emergent general understanding is that MASs, more than an effective technology, represent indeed a novel general-purpose paradigm for software development [99, 231]. Agent-based computing promotes designing and developing applications in terms of autonomous software entities (agents), situated in an environment (Section 2.3), and that can flexibly achieve their goals by interacting with one another in terms of high-level protocols and languages.

These features are well suited to tackle the complexity of developing software in modern scenarios: *(i)* the autonomy of application components reflects the intrinsically decentralised nature of modern distributed systems and can be considered as the natural extension to the notions of modularity and encapsulation for systems that are owned by different stakeholders; *(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 scenarios where software is expected to operate; *(iii)* the concept of agency provides for a unified view of artificial intelligence (AI) results and achievements, by making agents and MASs act as sound and manageable repositories of intelligent behaviours [230]. In

the last few years, together with the increasing acceptance of agent-based computing as a novel software engineering paradigm, there has been a great deal of research related to the identification and definition of suitable models and techniques to support the development of complex software systems (Subsection 2.1.1) in terms of MASs. This research endlessly proposes a variety of new metaphors, formal modelling approaches, development methodologies and modelling techniques, specifically suited to the agent-oriented paradigm.

In the reminder of this section the concepts of complex system and agent are illustrated respectively in Subsection 2.1.1 and Subsection 2.1.2, while the architecture of multi-agent systems is presented in Subsection 2.1.3.

2.1.1 Complex Systems

What are complex systems?

- A complex system is a system composed of interconnected parts that as a whole exhibit one or more properties (behaviour among the possible properties) not obvious from the properties of the individual parts (Wikipedia).
- A system comprised of a (usually large) number of (usually strongly) interacting entities, processes, or agents, the understanding of which requires the development, or the use of, new scientific tools, nonlinear models, out-of equilibrium descriptions and computer simulations [182].
- A system that can be analysed into many components having relatively many relations among them, so that the behaviour of each component depends on the behaviour of others [197].
- A system that involves numerous interacting agents whose aggregate behaviours are to be understood. Such aggregate activity is nonlinear, hence it cannot simply be derived from summation of individual components behaviour [132].

A complex system is any system featuring a large number of interacting components (agents, processes, etc.) whose aggregate activity is nonlinear (not derivable from the summations of the activity of individual components) and typically exhibits hierarchical self-organisation under selective pressures.

Almost all interesting processes in nature are highly cross linked. In many systems, however, a set of fundamental building blocks can be distinguished, which interact nonlinearly to form compound structures or functions with an identity that requires more explanatory devices than those used to explain the building blocks. This process of emergence of the need for new, complementary, modes of description is known as hierarchical self-organisation, and systems that observe this characteristic are defined as complex

[184]. Examples of these systems are gene networks that direct developmental processes, biological systems [130], social insect colonies, neural networks in the brain, social networks comprised of transportation, utilities, and telecommunication systems, as well as economies.

It has become clear in recent years, that the modelling of some phenomena, particularly, industrial and social phenomena, requires agents whose behaviour is not simply dictated by local, state-determined interaction: agents have access to knowledge which escapes local constraints (via communication) and is stored in media beyond the agent itself and its state. Indeed, many if not most researchers in Artificial Intelligence (AI), Cognitive Science and Psychology, have come to pursue the idea that intelligence is not solely an autonomous characteristic of agents, but heavily depends on social, linguistic, and organisational knowledge which exists beyond individual agents.

2.1.2 The Agents

At present, there is still a great deal of ongoing debate about exactly what constitutes an agent, yet there is nothing approaching a universal consensus. However, an increasing number of researchers find the following characterisation useful [98]:

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

This means that agents are:

- clearly identifiable problem solving entities with well-defined boundaries and interfaces,
- situated (embedded) in a particular environment – they receive inputs related to the state of that environment through their sensors and they act on the environment through their effectors,
- designed to fulfil a specific role – they have particular objectives to achieve, that can either be explicitly or implicitly represented within the agents,
- autonomous – they have control both over their internal state and over their own behaviour,
- capable of exhibiting flexible (context-dependent) problem solving behaviour: they need to be reactive (able to respond in a timely fashion to changes that occur in their environment in order to satisfy their design objectives) and proactive (able to opportunistically adopt new goals and take the initiative in order to satisfy their design objectives).

When adopting an agent-oriented view of the world, it soon becomes obvious that a single agent is insufficient. Most problems require or involve multiple agents: to represent the decentralised nature of the problem, the multiple loci of control, the multiple perspectives, or the competing interests [230]. Moreover, the agents will need to interact with one another, either to achieve their individual and social objectives or else to manage the dependencies that ensue from being situated in a common environment. These interactions range from simple semantic interoperation (the ability to exchange comprehensible communications), through traditional client-server type interactions (the ability to request that a particular action is performed), to rich social interactions (the ability to cooperate, coordinate and negotiate about a course of action). Whatever the nature of the social process, however, there are two points that qualitatively differentiate agent interactions from those that occur in other software engineering paradigms. Firstly, agent-oriented interactions generally occur through a high-level (declarative) agent communication language (typically based on speech act theory [62]). Consequently, interactions are usually conducted at the knowledge level: in terms of which goals should be followed, at what time, and by whom (cf. method invocation or function calls that operate at a purely syntactic level). Secondly, as agents are flexible problem solvers, operating in an environment over which they have only partial control and observability, interactions need to be handled in a similarly flexible manner. Thus, agents need the computational apparatus to make context-dependent decisions about the nature and scope of their interactions and to initiate (and respond to) interactions that were not necessarily foreseen at design time. In most cases, agents act to achieve objectives either on behalf of individuals/companies or as part of some wider problem solving initiative. Thus, when agents interact there is typically some underpinning organisational context. This context defines the nature of the relationship between the agents. For example, they may be peers working together in a team or one may be the boss of the others. In any case, this context influences an agent's behaviour. In many cases, relationships are subject to ongoing changes: social interaction means existing relationships evolve and new relations are created.

Although there are certain similarities between object- and agent-oriented approaches (both adhere to the principle of information hiding and recognise the importance of interactions), there are also a number of important differences [99]. First, objects are generally passive in nature: they need to be sent a message before they become active. Secondly, although objects encapsulate state and behaviour realisation, they do not encapsulate behaviour activation (action choice). Thus, any object can invoke any publicly accessible method on any other object. Once the method is invoked, the corresponding actions are performed. Additionally, object-orientation fails to provide an adequate set of concepts and mechanisms for modelling complex systems: for such systems “we find that objects, classes, and modules provide an essential yet insufficient means of abstraction” [12]. Individual objects represent too fine a granularity of behaviour and method invocation is too primitive as a mechanism for describing the types of interactions that take place. Recognition of these facts led to the development of more powerful abstraction mecha-

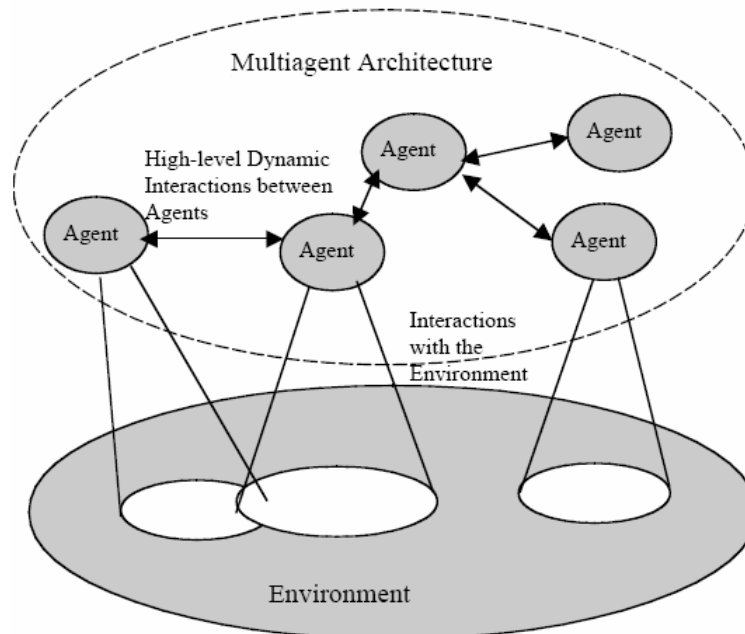


Figure 2.1: MAS architecture [230]

nisms such as design patterns, application frameworks, and componentware. Although these are undoubtedly a step forward, they fall short of the desired characteristics for complex system development. By their very nature, they focus on generic system functions and the mandated patterns of interaction are rigid and predetermined. Finally, object-oriented approaches provide only minimal support for specifying and managing organisational relationships (basically relationships are defined by static inheritance hierarchies). To summarise, the traditional view of an object-oriented system and view of an agent-oriented system have at least three distinctions [226]:

- agents embody a stronger notion of autonomy than objects, and in particular, they decide for themselves whether or not to perform an action on request from another agent;
- agents are capable of flexible (reactive, pro-active, social) behaviour, and the standard object model has nothing to say about such types of behaviour;
- a multi-agent system is inherently multi-threaded, in that each agent is assumed to have at least one thread of control.

2.1.3 The Multi-Agent Systems Architecture

Looking at the above definition of agent, it is clear that a MAS cannot be simply reduced to a group of interacting agents. Instead, the complete modelling of a MAS requires explicitly focusing also on the environment (Section 2.3) in which the MAS and its constituent agents are situated and on the society (Section 2.2) that a group of interacting agents give rise to. Modelling the environment implies identifying its basic features, the resources that can be found in the environment, and the way via which agents can interact with it. Modelling agent societies implies identifying the overall rules that should drive the expected evolution of the MAS and the various roles that agents can play in such a society. All the above considerations lead to the very general characterisation depicted in Figure 2.1, whose basic abstractions and overall architecture totally differ from that of traditional software engineering approaches (Figure 2.2).

Traditional object-based computing promotes a perspective of software components as “functional” or “service-oriented” entities that directly influences the way that software systems are architected [229]. Usually, the global design relies on a rather static architecture that derives from the decomposition (and modularisation) of the functionalities and data required by the system to achieve its global goals and on the definition of their interdependencies [196, 5]. In particular:

- objects are usually considered as service providers, responsible for specific portions of data and in charge of providing services to other objects (the “contractual” model of software development explicitly promotes this view);
- interactions between objects are usually an expression of inter-dependencies; two objects interact to access services and data that are not available locally;
- everything in a system tends to be modelled in terms of objects, and any distinction between active actors and passive resources is typically neglect.

In other words, object-oriented development, while promoting encapsulation of data and functionality and a functional-oriented concept of interactions, tends to neglect modelling and encapsulation of execution control. Some sort of “global control” over the activity of the system is usually assumed (e.g., the presence of a single execution flow or of a limited set of controllable and globally synchronised execution flows). However, assuming and/or enforcing such control may be not feasible in complex systems. Thus, rather than being at risk of losing control, a better solution would be to explicitly delegate control over the execution to the system components [231]—as in MASs. In fact:

- Delegating control to autonomous components can be considered as an additional dimension of modularity and encapsulation. When entities can encapsulate control in addition to data and algorithms, they can better handle the dynamics of a complex environment (local contingencies can be handled locally by components)

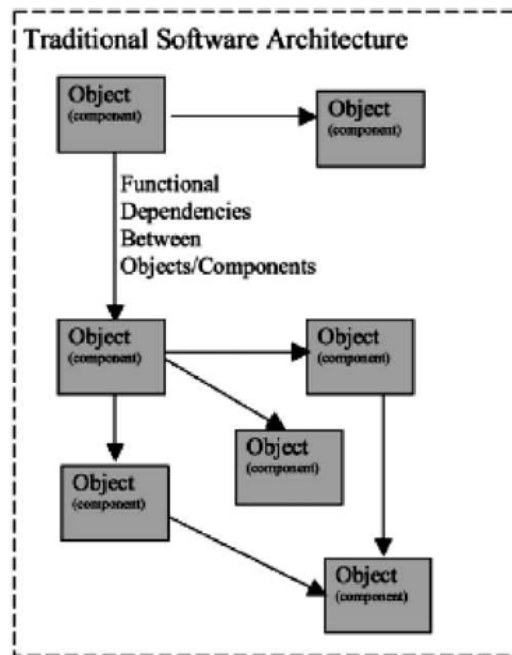


Figure 2.2: Traditional software architecture [230]

and can reduce their interdependencies (limiting the explicit transfer of execution activities). This leads to a sharper separation between the component-level (i.e., intra-agent) and system-level (i.e., inter-agent) design dimensions, in that also the control component is no longer global.

- The dynamics and openness of application scenarios can make it impossible to know a priori all potential interdependencies between components (e.g., what services are needed at a given point of the execution and with what other components to interact), as a functional-oriented perspective typically requires. Autonomous components delegated of their own control can be enriched with sophisticated social abilities, that is, the capability to make decisions about the scope and nature of their interactions at run-time and of initiating interactions in a flexible manner (e.g., by looking for and negotiating for service and data provision).
- For complex systems, a clear distinction between the active actors of the systems (autonomous and in charge of their own control) and the passive resources (passive objects without autonomous control) may provide a simplified modelling of the problem. In fact, the software components of an application often have a real-world counterpart that can be either active or passive and that, consequently, is better suited to being modelled in terms of both active entities (agents) and passive ones (environmental resources).

However, attempting to enrich more conventional approaches with novel features and characteristics to meet the novel needs is likely to introduce a dangerous mismatch between the abstraction level adopted and the conceptual level at which application problems have to be solved. Put simply, objects and components are too low level of abstraction for dealing with the complexity of today's software systems, and miss important concepts such as autonomy, task-orientation, situatedness and flexible interactions. For instance, object- and component-based approaches have nothing to say on the subject of designing negotiation algorithms to govern interactions, and do not offer insights into how to maintain a balance between reactive and proactive behaviour in a complex and dynamic situations. This forces applications to be built by adopting a functionally oriented perspective and, in turn, this leads to either rather static software architectures or to the need for complex infrastructures support to handle the dynamics and flexible reconfiguration and to support negotiation for resources and tasks.

In summary, agent-based computing promotes an abstraction level that is suitable for modern scenarios and that is appropriate for building flexible, highly modular, and robust systems, whatever the technology adopted to actually build the agents. This leads us to consider agent-based computing as a new software engineering paradigm (Chapter 3).

2.2 Societies

According to Castelfranchi [21, 24], sociality presupposes agents. At a very basic level, an agent is any entity able to act, i.e., to produce some causal effect and some change in its environment. Of course this broad notion is not enough for sociality. A more complex level of agenthood is needed. Agents are individual entities with social abilities. In general, they have a partial representation of the world around them, a limited ability to sense and change it, and typically rely on other agents for anything falling outside of their scope or reach (Figure 2.1). So, agents are to be thought as living dipped into societies: the behaviour of an individual agent is often not understandable outside its social structure. The behaviour of a buyer agent in an auction is difficult to be explained out of the context of the auction itself and of the rules that govern it. Dually, the behaviour of a society of agents cannot generally be expressed in terms of the behaviour of its composing agents. So, the rules governing an auction, in conjunction with the behaviour of the individual agents participating in it, lead to a global behaviour that could not be reduced to the mere composition of the individual's behaviour. Social rules harness agent interactions, and drive the global behaviour of a society towards the accomplishment of its global goals.

So, the most appealing metaphor for modelling and design MASs seems to be the *human society or (human organisation)* [229] in which

- A software system is conceived as the computational instantiation of a (possibly open) group of interacting and autonomous individuals (agents).

- Each agent can be seen as playing one or more specific roles: it has a well defined set of responsibilities or subgoals in the context of the overall system and is responsible for pursuing these autonomously. Such subgoals may be both altruistic (to contribute to a global application goal) or opportunistic (for an agent to pursue its own interests).
- Interactions are no longer merely an expression of interdependencies, and are rather seen as a means for an agent to accomplish its role in the system. Therefore, interactions are clearly identified and localised in the definition of the role itself, and they help to characterise the overall structure of the society and the position of the agent in it.
- The evolution of the activities in the society, deriving from the autonomous execution of agents and from their interactions, determines the achievement of the application goal, whether an a priori identified global goal (as, e.g., in a workflow management systems where altruistic agents contribute to the achievement of a specific cooperative project), or a goal related to the satisfaction of individual goals (as, for example, in agent-mediated auctions, whose purpose is to satisfy the needs of buyer and seller agents), or both (as, for example, in network enterprises exploiting market mechanisms to improve efficiency).

The organisational metaphor – other than being a natural one for human developers who are continuously immersed in a variety of organisational settings and opening up the possibility of reusing a variety of studies and experiences related to real-world organisations – appears to be appropriate for a wide range of software systems. On one hand, some systems are concerned with controlling and supporting the activities of some real-world organisation (e.g., manufacturing control systems, workflow management and enterprise information systems, and electronic marketplaces). Therefore, an organisation-based design may reduce the conceptual distance between the software system and the real-world system it has to support. On the other hand, other software systems, even if they are not associated with any pre-existing real-world organisation, may have to deal with problems for which human organisations could act as fruitful source of inspiration, having already shown to produce effective solutions (e.g., resource sharing, task assignment, and service negotiation).

More generally, whenever a software system is complex enough to warrant an agent-based approach and still requires a significant degree of predictability and reliability in all its parts, the organisational metaphor may be the most appropriate one. In fact, by relying on agents playing well-defined roles and interacting according to institutionalised patterns, the organisational metaphor promotes both micro-level (at the agents level) and macro-level (at the system level) control over the design and understanding of the overall system behaviour.

2.3 The Environment

Traditionally, the environment of a MAS is considered simply as the *deployment context* where agents are immersed—which includes e.g. the communication infrastructure, the network topology, the physical resources available. In this case, the environment of a MAS is basically considered as an output of system analysis, and designers are somehow passively subject to it.

On the other hand, today there is a growing recognition that the environment is a true design dimension of multi-agent applications [219]. It can encapsulate a significant portion of the system complexity, in terms of services, mechanisms and responsibilities that the agents can fruitfully be freed of. In [213, 219] a notion of environment is adopted that is not limited to the *external (MAS) environment*—the deployment context where the MAS has to work. Instead the focus is mostly on the *agent environment*, that is, the portion of the MAS that is external to the agents, but is anyhow part of the MAS and subject to the activity of MAS engineers.

Accordingly, as far as engineering is concerned, at least two main ingredients shaping the agent environment are recognised, which will be described in detail in this section: environment abstractions, entities of the environment encapsulating some functions; and topology abstractions, entities of MAS environment that represent the (either logical or physical) spatial structure.

2.3.1 Environment Abstractions

Following the work in [213], the agent abstraction is not the only one populating the MAS: rather, the MAS environment could be seen as decomposed in building blocks called *environment abstractions*. An environment abstraction is an entity of the MAS environment encapsulating some functions or services for the agents. An agent perceives the existence of such abstractions in the environment, has a (possibly implicit) awareness of the opportunities they provide, and accordingly interacts with them in order to achieve individual as well as social goals. From the development viewpoint, environment abstractions are seen as *loci* where the designer can enforce rules, norms, and functions, regulating the agent social behaviour.

The notion of environment abstraction is a means for interpreting a number of components and entities that populate MAS environment both at the conceptual and at the implementation levels, introduced by several MAS research works—all of them sharing the idea of being the target of agent interaction. Most existing infrastructures for agent environment are in fact seen as providing some incarnation of environment abstractions for the application at hand. Examples include *tuple centres* in TuCSoN [157], *co-fields* in TOTA [114], *shared virtual environments* of the automatic guided vehicles application in [222], *stigmergic fields* of the infrastructure for military operations in [192], *e-institution scenes* in AMELI [61], and so on.

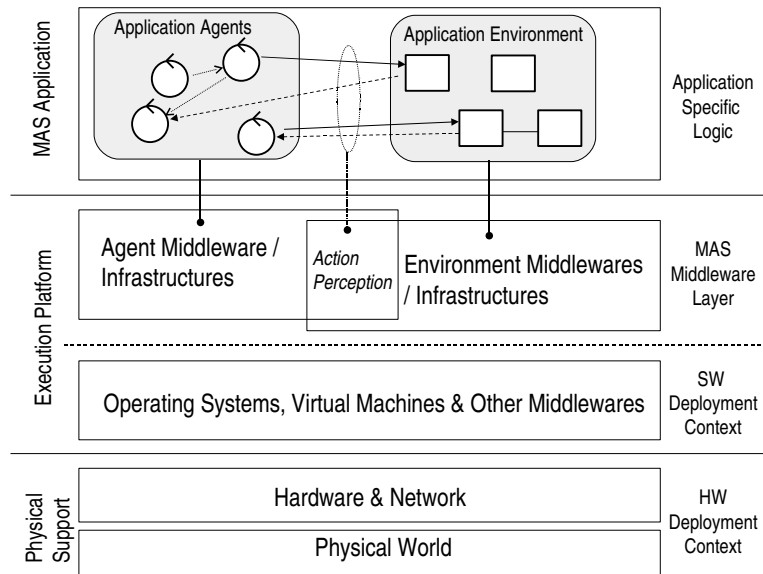


Figure 2.3: MAS layers with environment-based supports [213, 223]

Figure 2.3 provides a visualisation of how the concept of environment abstraction fits the different layers of a MAS—as discussed in detail in [213, 223]. At the lower layer the whole MAS (agents and environment) is immersed in a software and hardware deployment context; at the middleware layer, agent infrastructures (e.g. Jade [97]) are coupled with environment infrastructures—TOTA, TuCSon, and the other above mentioned ones. At the application layer, while the former provides the agent abstraction (circles in the picture) – agent life-cycle, and typically direct agent communication – the latter provides environment abstractions (boxes in the picture), with which agents interact by an action/perception mechanism resembling more closely physical action rather than communication [154].

2.3.2 Topology Abstractions

Besides considering the various abstractions populating the environment, it is also useful to consider its spatial structure, or *topology*. In particular, abstractions that support topology as a first-class notion – which is called *topology abstractions* – should be eventually considered part of MAS engineering. As a general definition, a topology is seen as a collection of neighbourhood sets, providing a structured notion of locality for the MAS, from local aspects up to the whole shape of the MAS. When considering a MAS as made of situated agents and environment abstractions, it is then natural to conceive MAS topology as a collection of sets of agents and environment abstractions. This affects notions like visibility among agents (whether an agent can see and communicate with

another one), visibility between agents and environment abstractions (whether an agent can see and interact with an environment abstraction), and agent mobility (which new neighborhood can be reached by an agent). This concept is rather important in MAS, because the visibility and accessibility of other entities strictly identifies which goals can be delegated (to other agents), and which services can be obtained (by environment abstractions), ultimately defining agent capability of achieving its objectives. In many cases, the concept of topology is tied with the physical structure of the deployment context, including both the network topology and the real physical environment – as in the case of robot environment – but in general it can suitably take into account a virtual notion of space.

As an example, among all the possible infrastructures there are two MAS infrastructures, **TOTA** [114] and **CArtAgO** [181], which explicitly deal with the notion of environment topology. In **TOTA** the environment is formed by a dynamic network of nodes. Other than agents, each site hosts one tuple space supporting the fabric of co-fields: a tuple can be cloned and moved from one space to another in the neighborhood. Accordingly, the topology abstraction is here implicitly defined by the neighborhood relation, and is made of the set of agents and tuple spaces in a given locality.

CArtAgO is a general-purpose infrastructure for environments, based on *artifacts* as an incarnation of the notion of environment abstraction—following the Agents & Artifacts meta-model (see Chapter 7). **CArtAgO** adopts the concept of *workspace* as a topology abstraction that can be used to define the topology of the computational environment. A workspace can be defined as an open set of artifacts and agents: artifacts can be dynamically added to or removed from workspaces, agents can dynamically enter or exit workspaces. A workspace is typically spread over the nodes of an underlying network infrastructure. Workspaces define topologies to structure agents' and artifacts' organisation and interaction, in particular workspaces are used as scopes for event generation and perception.

It is clear that even though environment abstractions, topology abstractions, and the interplay between them can be considered as key points in the modelling and design of the environment, existing MAS approaches generally do not explicitly take into account either of them [4]. This happens in spite of the fact that a large class of problems is characterised by unavoidable spatial features—several domains deal with space itself (e.g. geographical location) or a model of it (e.g. information flow in an organisational structure)—and by non-autonomous computational entities in the environment playing a significant role in the overall MAS goals.

3

From Software Engineering to Agent Oriented Software Engineering

This chapter deals with the shift from traditional software engineering to agent-oriented software engineering. The researches in the area of AOSE mainly focus on proposing methodologies for agent systems, i.e., on identifying the guidelines to drive the various phases of agent-based software development and the abstractions to be exploited in these phases. However, very little attention has been paid so far to the basic issue of engineering the process subjacent the development activity, i.e., of disciplining the execution of the different phases involved in the software lifecycle. Since these aspects are related to the software engineering in general independently from the specific building blocks adopted for modelling the systems, the definition of these key concepts seems very useful in order to allow readers to better understand the AOSE field – and the remainder of this thesis – specially because for the meaning of some of these concepts there is no agreement in the scientific community.

So, this chapter is organised as follows. Section 3.1 introduces a general view about the software and its characteristics, while Section 3.2 introduces the traditional software engineering and defines the concepts of software development process (Subsection 3.2.1), methodology (Subsection 3.2.2) and tool (Subsection 3.2.3). Finally Section 3.3 introduces the general concepts related to the AOSE illustrating the paradigm shift (Subsection 3.3.1) and the main AOSE ingredients (Subsection 3.3.2), and Section 3.4 propose a summary of this chapter.

3.1 Software

Software is abstract and intangible [199], it is not constrained by materials, or governed by physical laws or by manufacturing process. This simplifies software engineering as there are no physical limitations on the potential of software. However, this lack of natural constraints means that software can easily become extremely complex and hence very difficult to understand. So, software engineers should adopt a *systematic and organised approach* to their work and *use appropriate tools and techniques* depending on the problem

<i>Characteristic</i>	<i>Description</i>
Maintainability	software should be written in such a way that it may evolve to meet the changing needs of customers. This is a critical attribute because software change is an inevitable consequence of changing business environment
Dependability	software dependability has a range of characteristics, including reliability, security and safety. Dependable software should not cause physical or economic damage in the event of system failure
Efficiency	software should not make wasteful use of system resources such as memory and processor cycles. Efficiency therefore includes responsiveness, processing time, memory utilisation, etc. . .
Usability	Software must be usable, without undue effort, by the type of user for whom it is designed. This means that it should have an appropriate user interface and adequate documentation

Figure 3.1: Essential attributes of good software

to be solved, the development constraints and the resources available.

As well as the services that they provide, software products have a number of associated attributes that reflect the quality of that software. These attributes are not directly concerned with what software does. Rather, they reflect its behaviour while it is executing and the structure and the organisation of the source program and associated documentation. Example of these *non-functional* attributes are the software's response time to a user query and the understandability of the program code.

The specific set of attributes that might be expected from a software system obviously depends on its applications. Therefore, a banking system must be secure, an interactive game must be responsive, a telephone switch must be reliable, and so on. These can be generalised into a set of attributes show in Figure 3.1, which are essential characteristics of a well-designed software system [199].

3.2 Traditional Software Engineering

Sommerville [199] defines Software Engineering (SE) as

an engineering discipline that is concerned with all aspects of software production from the early stage of system specification to maintaining the system after it has gone into use.

In this definition there are two key phrases:

1. Engineering discipline — Engineers make things work. They apply theories, methodologies and tools where these are appropriate, but they use them selectively and always try to discover solutions to problems even when there are no applicable theories and methodologies. Engineers also recognise that they must work to organisational and financial constraints, so they look for solutions within these constraints.
2. All aspects of software production — SE is not just concerned with the technical process of software development but also with the development of *tools*, *methodologies* and theories to support software production.

In general, software engineers adopt a systematic and organised approach to their work, as this is often the most effective way to produce high-quality software. However, engineering is all about selecting the most appropriate methodology for a set of circumstances and more creative, less formal approach to development may be effective circumstances.

The first key challenge in the SE is the production of software with a well-defined quality level, but this is not enough. There is also a need to model and engineer the development process that should be controllable, and well documented. These needs require *abstractions*, *process*, *methodologies* and *tools*.

Software deals with *entities* having a real-world counterpart such as numbers, dates, names, persons, documents, etc. . . . These entities must be modelled in terms of *abstractions*. Abstractions are the building blocks for creating the models and they depend on the available technologies: for example functions, objects, agents, etc. . . . Different kinds of abstraction lead both to different ways of *thinking* about the problems and the solutions, and to different levels of *model complexity*: complicated problems are well captured by object and agent abstractions, while functions could lead to have very very complex models for representing the problem. In the same way the models of the solution are heavily influenced by the paradigm.

In the reminder of this section are presented the definitions of software development processes (Subsection 3.2.1) and methodologies (Subsection 3.2.2), while Subsection 3.2.3 presents tools for SE.

3.2.1 Software Development Processes

A Development Process [26] is an ordered set of steps that involve all those activities, constraints and resources required to produce a specific desired output satisfying a set of input requirements. Typically, a process is composed by different stages/phases put in relation with each other. Each stage/ phase of a process identifies a portion of work to be done in the context of the process, the resources to be exploited to that purpose and the constraints to be obeyed in the execution of the phase. Case by case, the work in a phase can be very small or more demanding. Phases are usually composed by a set of activities that may, in turn, be conceived in terms of smaller atomic units of work (steps).

A *Software Development Process* [67] is the coherent set of policies, organisational structures, technologies, procedures and deliverables that are needed to conceive, develop, deploy and maintain a software product. A software process exploits a number of contributions and concepts [67]:

- *Software development technology*: technological support used in the process. Certainly, to accomplish software development activities there is the need for tools, infrastructures, and environments.
- *Software development methods and techniques*: guidelines on how to use technology and accomplish software development activities. The methodological support is essential to exploit technology effectively.
- *Organisational behaviour*: the science of organisations and people.
- *Marketing and economy*: software development is not a self-contained endeavor. As any other product, software must address real customers' needs in specific market settings.

There are four fundamental process activities that are common in all software processes. These are [199]:

- *Specification* — is the process of understanding and defining what services are required from the system and identifying the constraints on the system's operation and development.
- *Design and Implementation* — is the process of description of the structure of the software to be implemented, the data which is part of the system, the interface between the system's components and, sometimes, the algorithms used. Subsequently this specification is converted into an executable system.
- *Validation* — is the process that intended to show that a system conforms to its specification and that the system meets the expectations of the customers.
- *Evolution* — is the process that changes the software in response to changing demands.

There is not an ideal process. Different types of systems need different development processes [199]: for example in real time software, an aircraft has to be completely specified before development begins, while in e-commerce systems, the specification and the program are usually developed together. Consequently, the generic activities, specified above, may be organised in different ways and described at different levels of detail for different types of software. The use of an inappropriate software process may reduce the quality or the usefulness of the software product to be developed and/or increased.

A *Software Process Model* is a simplified representation of a software process, presented from a specific perspective [199]. Process model prescribes around which phases a process should be organised, in which order such phases should be executed, and when interactions and coordination between the work of the different phases should occur. In other words, a process model defines a skeleton, a template, around which to organise and detail an actual process. Examples of process models are:

- Workflow model — this shows the sequence of activities along with their inputs, outputs and dependencies
- Activity model — this represents the process as a set of activities, each of which carries out some data transformation
- Role/action model — this depicts the roles of the people involved in the software process and the activities for which they are responsible

The best known generic process models are:

- *Waterfall* — separate and distinct phases of specification and development
- *Iterative development* — specification, development and validation are interleaved
- *Component-based software engineering* — the system is assembled from existing components

3.2.2 Methodologies

Disagreement exists regarding the relationship between the terms *method* and *methodology*. In common use, the methodology is frequently substituted for method; seldom does the opposite occur. Some argue this occurs because methodology sounds more scholarly or important than method. A footnote to the word methodology in the 2006 American Heritage Dictionary notes that

the misuse of methodology obscures an important conceptual distinction between the tools of scientific investigation (properly methods) and the principles that determine how such tools are deployed and interpreted (properly methodologies).

In Software Engineering there is no common denominator: on one hand some authors argue that a software engineering method is a recipe, a series of steps, to build software, while a methodology is a codified set of recommended practices. In this way, a software engineering method could be part of a methodology. On the other hand, some authors believe that in a methodology there is an overall philosophical approach to the problem. Using these definitions, Software Engineering is rich in methods, but has fewer methodologies.

The definitions adopted in this thesis for methodology and method are from Ghezzi and Cernuzzi. In particular, Ghezzi et al. [73] define *methodology* as a collection of methods covering and connecting different stages in a process. The purpose of a methodology is to prescribe a certain coherent approach for solving a problem in the context of a software process by preselecting and putting in relation a number of methods. A methodology has two important components: one that describes the process elements of the approach, and a second that focuses on the work products and their documentation. According to this definition of methodology, Cernuzzi et al. [26] define a *method* as a way of performing some kind of activity within a process, in order to properly produce a specific output (i.e., an model or a document) starting from a specific input (again, an artefact or a document). Any phases of a process, to be successfully applicable, should be complemented by some methodological guidelines (including the identification of the techniques and tools to be used, and the definition of how artifacts have be produced) that could help the involved stakeholders in accomplishing their work according to some defined best practices.

Based on the above definitions, and comparing software processes and methodologies, some common elements can be found in their scope [26]:

- both focus on what we have to do in the different activities needed to construct a software system
- however, while the software development process is more centered on the global process including all the stages, their order and time scheduling, the methodology focuses more directly on the specific techniques to be used and work to be produced

In this sense, methodologies focus more explicitly on *how* to perform the activity or tasks in some specific stages of the process, while processes may also cover more general management aspects, e.g., basic questions about *who* and *when*, and *how much*.

As for the software development process, there is not an ideal methodology, and different methodologies have different areas where they are applicable. For example, object-oriented methodologies are often appropriate for interactive systems but not for systems' stringent real-time requirements [199].

3.2.3 Tools

The acronym CASE stands for Computer-Aided Software Engineering. It covers a wide range of different types of programs that are used to support software process activities such as requirement analysis, system modelling, debugging and testing. All methodologies should come with associated CASE technology such as editors for the notation used in the methodologies, analysis modules which check the system model according to the methodology rules and report generators to help create system documentation. The CASE tools may also include a code generator that automatically generates source code from the system model and some process guidance for software engineers.

3.3 Agent Oriented Software Engineering

Traditional SE abstractions fall short when applied to areas of growing relevance such as electronic commerce, enterprise resource planning, and mobile computing: such areas, in fact, generally call for complex systems and open architectures that may evolve dynamically over time so as to accommodate new components and meet new requirements. This is probably one of the main reasons why the *agent* metaphor and the agent-oriented paradigm are gaining momentum in these areas.

Today's software engineering approaches are increasingly adopting abstractions approaching that of agent-based computing. This trend can be better understood by recognising that the vast majority of modern distributed systems scenarios are intrinsically prone to be developed in terms of MASs, and that modern distributed systems are already de facto MASs, i.e., they are indeed composed of autonomous, situated, and social components [230]. As far as autonomy is concerned, almost all of today's software systems already integrate autonomous components. At its weakest, autonomy reduces to the ability of a component to react to and handle events, as in the case of graphical interfaces or simple embedded sensors. However, in many cases, autonomy implies that a component integrates an autonomous thread of execution, and can execute in a proactive way. This is the case of most modern control systems for physical domains, in which control is not simply reactive but proactive, implemented via a set of cooperative autonomous processes or, as is often the case, via embedded computer-based systems interacting with each other or via distributed sensor networks. The integration in complex distributed applications and systems of (software running on) mobile devices can be tackled only by modelling them in terms of autonomous software components. Internet based distributed applications are typically made up of autonomous processes, possibly executing on different nodes, and cooperating with each other – a choice driven by conceptual simplicity and by decentralised management rather than by the actual request for autonomous concurrent activities.

Today's computing systems are also typically situated. That is, they have an explicit notion of the environment where components are allocated and executed, and with which components explicitly interact. Control systems for physical domains, as well as sensor networks, tend to be built by explicitly managing data from the surrounding physical environment, and by explicitly taking into account the unpredictable dynamics of the environment via specific event-handling policies. Mobile and pervasive computing applications recognise (under the general term of context-awareness) the need for applications to model explicitly environmental characteristics and environmental data rather than to model them implicitly in terms of internal object attributes. Internet applications and web-based systems, to dive into the existing Internet environment, are typically engineered by clearly defining the boundaries of the system in terms of the “application”, including the new application components to be developed, and “middleware” level, as the environmental substrate in which components are to be embedded.

Sociality in modern distributed systems comes in different flavors: (i) the capability of components to support dynamic interactions, i.e., interaction established at run-time with previously unknown components; (ii) the somewhat higher interaction level, overcoming the traditional client-server scheme; (iii) the enforcement of some sorts of societal rules governing the interactions. Control systems for critical physical domains typically run forever, cannot be stopped, and sometimes cannot even be removed from the environment in which they are embedded. Nevertheless, these systems need to be continuously updated, and the environment in which they live is likely to change frequently, with the addition of new physical components and, consequently, of new software components and software systems. For all these systems, managing openness and the capability to automatically re-organise interaction patterns is crucial, as is the ability of a component to enter new execution contexts in respect of the rules that are expected to drive the whole execution of the system. With reference to pervasive computing systems, lack of resources, power, or simply communication unreachability, can make nodes come and go in unpredictable ways, calling for re-structuring of communication patterns, as well as for high-level negotiations for resource provision. Such issues are even exacerbated in mobile networking and P2P systems, where interactions must be made fruitful and controllable despite the lack of any intrinsic structure and dynamics of connectivity. Similar considerations apply to Internet-based and open distributed computing. There, software services must survive the dynamics and uncertainty of the Internet, must be able to serve any client component, and must also be able to enact security and resource control policy in their local context, e.g., a given administrative domain. E-marketplaces are the most typical examples of this class of open Internet applications. The explicit adoption of agent-based concepts in systems engineering would carry several advantages:

- autonomy of application components, even if sometimes directly forced by the distributed characteristic of the operational environment, enforces a stronger notion of encapsulation (i.e., encapsulation of control rather than of data and algorithms), which reduces the complexity of managing systems with a high and dynamically varying number of components;
- taking into account situatedness explicitly, and modelling environmental resources and active computational entities in a differentiated way, rather than being the recognition of a matter of fact, provides for a better separation of concerns which, in turn, help to reduce complexity;
- dealing with dynamic and high-level interactions (i.e., with societal rather than with architectural concepts) enables to address in a more flexible and structured way the intrinsic dynamics and uncertainties of modern distributed scenarios.

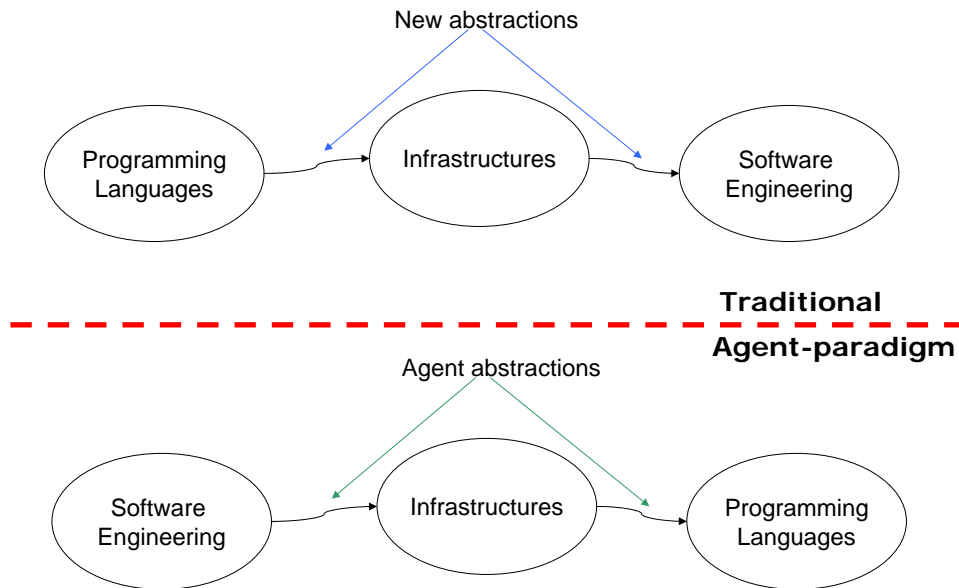


Figure 3.2: Paradigms shift in traditional SE (top) and AOSE (bottom)

3.3.1 Paradigm Shift

A change of paradigm is always a dramatic event in any scientific and engineering field. The rapid paradigm shift dropped technology behind: while in the past new abstractions used to come from programming languages, and were later included in software engineering practice, now it is often the case that technologies adopted for MAS development and deployment do not support the novel abstractions adopted in the AOSE analysis and design phases (Figure 3.2). Such a gap mainly depends on AO methodologies and AO infrastructures having evolved along two parallel, yet somehow inverse, paths: a top-down evolution for AO methodologies, a bottom-up path for AO infrastructures. In fact, on the one side, abstractions and metaphors (models and structures) from human organisations have been used to analyse, model and design software systems. There, modelling *agent societies* means to identify the rules – global or local – that should drive the expected MAS evolution, and the roles that agents should play. On the other side, AO infrastructures have typically evolved out from existing (mainly, object-oriented) programming languages and development environments, “stretching” the agent paradigm on top of more traditional paradigms and technologies [155].

So a significant part of the work in SE is finding the *right models and abstractions* for addressing the new systems issues. In general, there will be multiple candidates and the difficult task is picking the most appropriate one. When designing software, the most powerful abstractions are those that *minimise* the semantic gap between the units of analysis that are intuitively used to conceptualise the problem and the constructs present

in the solution paradigm.

In order to minimise this gap, complex systems such as MAS could be characterised in terms of sub-systems, sub-system components, interactions and organisational relationships. Sub-systems naturally correspond to agent organisations. They involve a number of constituent components that act and interact according to their role within the larger enterprise. The interplay between the sub-systems and between their constituent components is most naturally viewed in terms of high-level social interactions: “at any given level of abstraction, we find meaningful collections of entities that collaborate to achieve some higher level view” [12]. This view accords precisely with the knowledge-level treatment of interaction afforded by the agent-oriented approach. Multi-agent systems are invariably described in terms of “cooperating to achieve common objectives”, “coordinating their actions” or “negotiating to resolve conflicts”. Complex systems involve changing webs of relationships between their various components. They also require collections of components to be treated as a single conceptual unit when viewed from a different level of abstraction. Here again the agent-oriented mindset provides suitable abstractions. A rich set of structures is typically available for explicitly representing and managing organisational relationships. Interaction protocols exist for forming new grouping and disbanding unwanted ones. Finally, structures are available for modelling collectives. The latter point is especially useful in relation to representing sub-systems since they are nothing more than a team of components working to achieve a collective goal.

Organisational constructs are first-class entities in agent systems. Thus explicit representations are made of organisational relationships and structures [98]. Moreover, agent-based systems have the concomitant computational mechanisms for flexibly forming, maintaining and disbanding organisations. This representational power enables agent-oriented systems to exploit two facets of the nature of complex systems. Firstly, the notion of a primitive component can be varied according to the needs of the observer. Thus at one level, entire sub-systems can be viewed as a singleton, alternatively teams or collections of agents can be viewed as primitive components, and so on until the system eventually bottoms out. Secondly, such structures provide a variety of stable intermediate forms, that, as already indicated, are essential for rapid development of complex systems. Their availability means that individual agents or organisational groupings can be developed in relative isolation and then added into the system in an incremental manner. This, in turn, ensures there is a smooth growth in functionality.

3.3.2 AOSE Ingredients

As far as software engineering is concerned, the key implication is that the design and development of software systems according to a (new) paradigm can by no means rely on conceptual tools and methodologies conceived for a totally different (old) paradigm [230]. Even if it is still possible to develop a complex distributed system in terms of objects and client-server interactions, such a choice appears odd and complicated when the system

is a MAS or it can be assimilated to a MAS. Rather, a brand new set of conceptual and practical tools – specifically suited to the abstractions of agent-based computing – is needed to facilitate, promote, and support the development of MASs, and to fulfil the great general-purpose potential of agent-based computing. A short summary of the AOSE ingredients is worth reporting [230].

Agent modelling. Novel formal and practical approaches to component modelling are required, to deal with autonomy, pro-activity, and situatedness. A variety of agent architectures is being investigated, each of which is suitable to model different types of agents or specific aspects of agents: purely reactive agents, logic agents, agents based on belief, desire and intentions. Overall, this research has so far notably clarified the very concept of agency and its different facets.

Society modelling. The complexity of systems no longer allows any system components to be completely controlled/designed/governed merely as individuals. Correspondingly, intelligence embedded within agents is often not enough to build up intelligent systems. The very notion of situated intelligence, when seen through the eyes of intelligent systems engineers, calls for a suitable design of what is outside the agents: societies that agents form and where they get deployed, and environments where agents live. So, the design of intelligent systems seems to require: on the one hand, suitable design abstractions to support social intelligence, i.e., intelligence exhibited by agent societies, which cannot directly be ascribed to individual intelligent (component) agents; on the other hand, suitable infrastructures shaping the agent environment so as to fully enable and promote the exploitation of both individual and social intelligence. The former requirement clearly emerges from several AO methodologies, adopting organisational models to describe and design systems in terms of organisational structure (roles involved), organisational patterns (roles relationships), and organisational rules (constraints on roles and their interactions).

MAS architectures. As it is necessary to develop new ways of modelling the components of a MAS, in the same way it is necessary to develop new ways of modelling a MAS as a whole. Detaching from traditional functional-oriented perspectives, a variety of approaches are being investigated to model MASs. In particular, approaches inspired by societal, organisational, and biological metaphors, are the subject of the majority of researches and are already showing the specific suitability of the different metaphors in different application areas.

AO methodologies. Traditional methodologies of software development, driving engineers from analysis to design and development, must be tuned to match the abstractions of agent-oriented computing. To this end, a variety of novel methodologies to discipline and support the development process of a MAS have been defined in the past few years (Chapter 4), clarifying the various sets of abstractions that must come into play during MAS development and the duties and responsibilities of software engineers. The defini-

tion of agent-specific methodologies is definitely one of the most explored topics in AOSE, and a large number of AO methodologies – describing how the process of building a MAS should/could be organised – has been proposed in the literature (Chapter 4). However, what characterises most of the methodologies proposed so far is that they assume a very traditional process models [199] (from analysis to design, implementation, and maintenance) for organising the process of building a MAS. It appears rather odd that most proposals for AO methodologies adopt a standard process model when, in the real world of industrial software development, such a standard model is rarely applied. It is a matter of fact that, in many cases, software is developed following a non-structured process: analysis, design, and implementation, often collapse into the frenetic work of a bunch of technicians and programmers, directly interacting with clients (to refine typically vague specifications), and striving to deliver the work on time [230]. In the mainstream community of SE, such a situation is getting properly attributed via the definition of novel software process models, specifically conceived to give some flavor of “engineering” to such chaotic and frenetic processes (e.g., agile and extreme software process models, and method engineering [15, 16, 84]). In the area of AOSE, a similar direction should be explored too, possibly exploiting the fact that the very abstractions of agents may promote the identification of different and more agile process models and method engineering [36].

Meta-models. AO methodologies, as well as non-AO ones typically start from a meta-model, identifying the basic abstractions to be exploited in development (e.g., agents, roles, environment, organisational structures). On this base, they exploit and organise these abstractions so as to define guidelines on how to proceed in the analysis, design, and development, and on what output to produce at each stage. Actually, several works [86, 10] are focusing on the identification of appropriate meta-models for AO methodologies and process models, where a meta-model is intended as rational analysis and identification of the abstractions used in MAS development. Those efforts aim at unifying the different abstractions adopted in existing methodologies and the process models, and also at identifying which relationships may exist among them. This may be used to better understand the real usefulness of the abstractions and to improve or unify processes and methodologies. Also, those effort may help researchers and practitioners to identify and develop conceptual instruments and practical tools for an efficient processes management (see Chapter 5 and 6).

Notation techniques. The development of specific notation techniques to express the outcome of the various phases of a MAS development process are needed, because traditional object- and component-oriented notation techniques cannot easily apply. In this context, the AUML proposal [64], extending standard UML toward agent-oriented systems, is the subject of a great deal of research and it is rapidly becoming a de facto standard: newly proposed AO methodologies tend to adopt AUML as the basic notation technique and newly proposed interaction patterns in a variety of applications are usu-

ally expressed in terms of AUML diagram. Despite the current enthusiasm for AUML, AUML seems not be the ultimate answer. Beside the current period of transition, in which AUML will play an important role, the complexity, dynamics, and situated nature of modern software systems cannot be effectively dealt with by notations and modelling techniques originated for static and not situated software architectures. Whatever extensions will be proposed for AUML, they will intrinsically carry on the original shortcomings of the original object-oriented proposal. In the traditional software engineering community, the shortcomings of standard UML are becoming evident, and novel notations are being explored to account for higher dynamics and complexities. Accordingly, a great challenge in the area of AOSE will be that of identifying brand new notations and modelling techniques, conceived from scratch to suit the specific characteristics of MASs. Together with AUML, novel proposals should be very welcome and not simply discarded because they do not conform to widespread standards. The fact that reasonable proposals in this direction can actually be formulated and effectively compete with AUML is witnessed by, e.g., the work of Sturm et al. [201], describing a modified version of OPM [53] specifically suited for MASs.

AO infrastructures. To support the development and execution of MASs, novel tools and novel software infrastructures are needed. In this context, various tools are being proposed to transform standard MAS specifications (i.e., AUML specifications) into actual agent code, and a variety of infrastructures have been deployed to provide proper services supporting the execution of distributed MASs.

3.4 Summing up

Agents and multi-agent systems, other than a technology, represent a brand new paradigm for software development [99, 229]. When adopting agents as the basic conceptual components of software systems, software has to be conceived in terms of autonomous task-oriented entities, interacting with each other in a high-level way, leading to possibly very articulated organisations. Roughly speaking, this represents “only” a paradigm shift: the building blocks – e.g. functions, object, agent – are changed according to the specific paradigm adopted, but the key concepts underpinning the software engineering – processes, method and methodology – should be the same.

Obviously this is an high level viewpoint, in fact the paradigms adopted lead to different levels of *model complexity*: complicated problems are well captured by objects and agents, while functions could lead to have very complex models for representing the problem. In the same way the models of the solution are heavily influenced by the paradigm, so the construction of processes and methodologies is deeply influenced by the paradigm adopted. However, the key definitions of processes and methodologies presented in this chapter should be never forgotten when a new paradigm and its specialised software engineer field are introduced.

4

Agent Oriented Methodologies

This chapter presents an overview of the most known AO methodologies. In particular Section 4.1 presents an introduction about the methodologies for agent-oriented systems and their roots, while Section 4.2 presents the most known AO methodologies. Finally Section 4.3 presents a discussion about methodologies.

4.1 Methodologies for Agent-Oriented Systems

Agent-oriented (AO) methodologies suggest a clean and disciplined approach to analyse, design and develop multi-agent systems, using specific methods and techniques.

While there is a much debate on the use of terminology in various subcultures of information systems and SE, it can be generally agreed that a methodology has two important components: one that describes the process elements of the approach, and a second that focuses on the work products and their documentation [186]. The second of these is more visible in the usage of a methodology, which is why the object-oriented (OO) modelling language UML [143] is so frequently (totally incorrectly) equated with “all things OO” or even described as a methodology. Instead, in the AO methodologies world are adopted several notations: someone uses UML or its agent-focused counterpart AUML [64], while others eschew this as being inadequate to support the concepts of agents introducing instead their own individualistic notation and underpinning concepts. This is a consequence of that the existing methodologies have several roots. Some are based on ideas from artificial intelligence (AI), others as direct extensions of existing OO methodologies, whilst yet others try and merge the two approaches by taking a more purist approach yet allowing OO ideas when these seem to be sufficient (Figure 4.1).

Several methodologies acknowledge a direct descentance from full OO methodologies. In particular, MaSE [224] acknowledges influences from Kendall et al. [104] as well as an heredity from AAIL [106] which in turns was strongly influenced by the OO methodology of Rumbaugh and colleagues called OMT [189]. Similarly, the OO methodology of Fusion [32] was said to be highly influential in the design of Gaia [225, 229]. Two other OO approaches have also been used as the basis for AO extensions. RUP [110] has formed the basis for ADELFE [173] and also for MESSAGE [69], which, in turn, is the basis for IN-

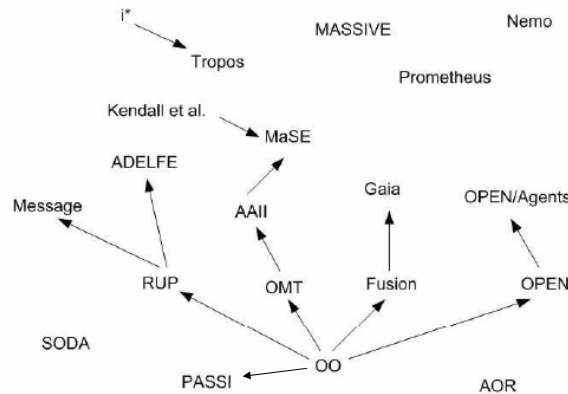


Figure 4.1: Influences of OO methodologies on AO methodologies [87]

GENIAS [170]. More recently, RUP has also been used as one of the inputs (together with AOR [217]) for RAP [208]. Secondly, the OPEN approach to OO software development has been extended significantly to support agents, sometimes called Agent OPEN [160]. Finally, two other methodologies exhibit influences from object-oriented methodological approaches. Prometheus [163], although not an OO descendant, does suggest using OO diagrams and concepts whenever they exist and are compatible with the agent-oriented paradigm. Similarly, PASSI [35] merges OO and MAS ideas, using UML as its main notation.

Somewhat different is the MAS-CommonKADS methodology [95]. This is a solidly-AI-based methodology that claims to have been strongly influenced by OO methodologies, notably OMT. Then there are the methodologies that do not acknowledge any direct genealogical link to other approaches, OO or AO. Tropos [13] has a significant input from i^* ([227]) and a distinct strength in early requirements modelling, focussing as it does on describing the goals of stakeholders that describe the “why” as well as the more standard support for “what” and “how”. This use in Tropos of the i^* modelling language (particularly in the analysis and design phases) gives it a different look and feel from those that use Agent UML (a.k.a. AUML [64]) as a notation. It also means that the non-OO mindset permits users of Tropos to take a unique approach to the modelling of agents in the methodological context. Another example is SODA [3], which focuses exclusively on the engineering of the interaction space of complex systems using the agent paradigm, providing direct support at the analysis and design stage for describing social aspects and MAS environment.

Other approaches include Nemo [92], MASSIVE [112], Cassiopeia [33] and CAMLE [195]—although in CAMLE there are some parallels drawn between its notion of “caste” and the concept of an OO class as well as some connection to UML’s composition and aggregation relationships. Further comparisons of these methodologies are undertaken in Tran and Low [140], which complements and extends earlier framework-based evaluative

studies of, for instance, Cernuzzi and Rossi [27], Dam and Winikoff [39], and Sturm and Shehory [203].

4.2 The most known AO Methodologies

This section shows an overview of the best known AO methodologies. In particular Subsection 4.2.1 presents the Gaia methodology, Subsection 4.2.2 presents the ADELFE methodology, Subsection 4.2.3 presents the Tropos methodology, Subsection 4.2.4 presents the PASSI methodology, Subsection 4.2.5 presents the MaSE methodology, Subsection 4.2.6 presents the INGENIAS methodology, Subsection 4.2.7 presents MESSAGE, and finally Subsection 4.2.8 presents Prometheus.

4.2.1 Gaia

Gaia [228, 229] (Figure 4.2) focuses on the use of organisational abstractions to drive the analysis and design of MASs (see also Section 6.4). Gaia models both the macro (social) aspect and the micro (agent internals) aspect of a MAS, and devotes a specific effort to model the organisational structure and the organisational rules that govern the global behaviour of the agents in the organisation.

The goal of the analysis phase in Gaia, covering the requirements in term of functions and activities, is to firstly identify which loosely coupled sub-organisations possibly compose the whole system and then, for each of these, produce four basic abstract models: *(i)* the environmental model, to capture the characteristics of the MAS operational environment; *(ii)* a preliminary roles model, to capture the key task-oriented activities to be played in the MAS; *(iii)* a preliminary interactions model, to capture basic interdependencies between roles; and *(iv)* a set of organisational rules, expressing global constraints/directives that must underlie the MAS functioning.

The above analysis models are used as input to the architectural design phase. In particular, the architectural design phase is in charge of defining the most proper organisational structure for the MAS, i.e., the topology of interactions in the MAS and the control regime for the MAS which most effectively enables the fulfillment of the MAS goals. The definition of the organisational structure has to account for a variety of factors, including the need for somewhat reflecting the structure of the real-world organisation in the MAS structure, the characteristics of the environment and of the patterns of access to it, the need for simplifying the enactment of the organisational rules, the need to respect any identified non-functional requirement, as well as the obvious need to keep the design as simple as possible. Once the most appropriate organisational structure is defined, the roles and interactions models identified in the analysis phase (which were preliminary, in that they were not situated in any actual organisational structure) can be finalised, to account for all newly identified interactions and possibly for newly identified roles.

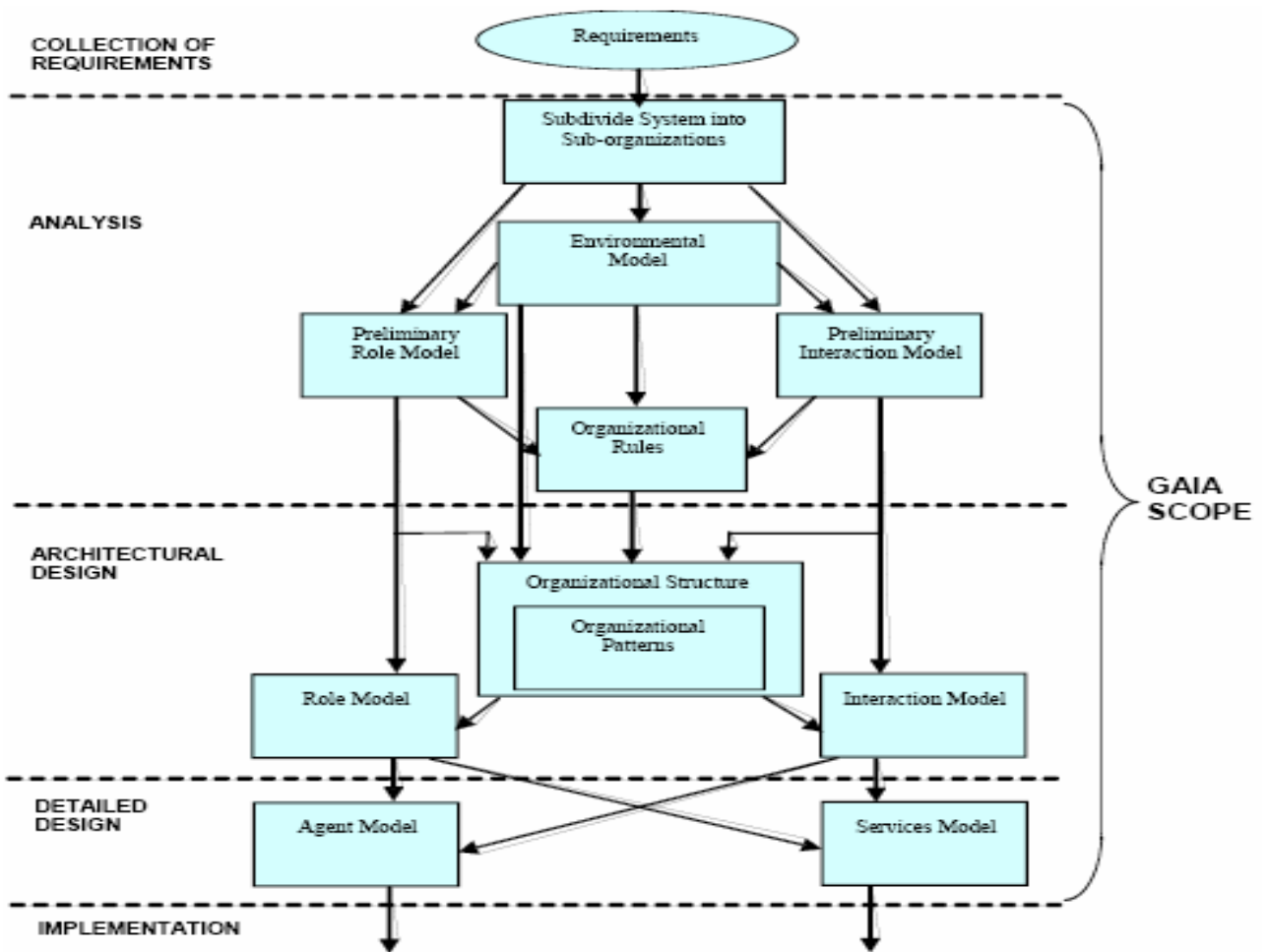


Figure 4.2: An overview of Gaia [229]

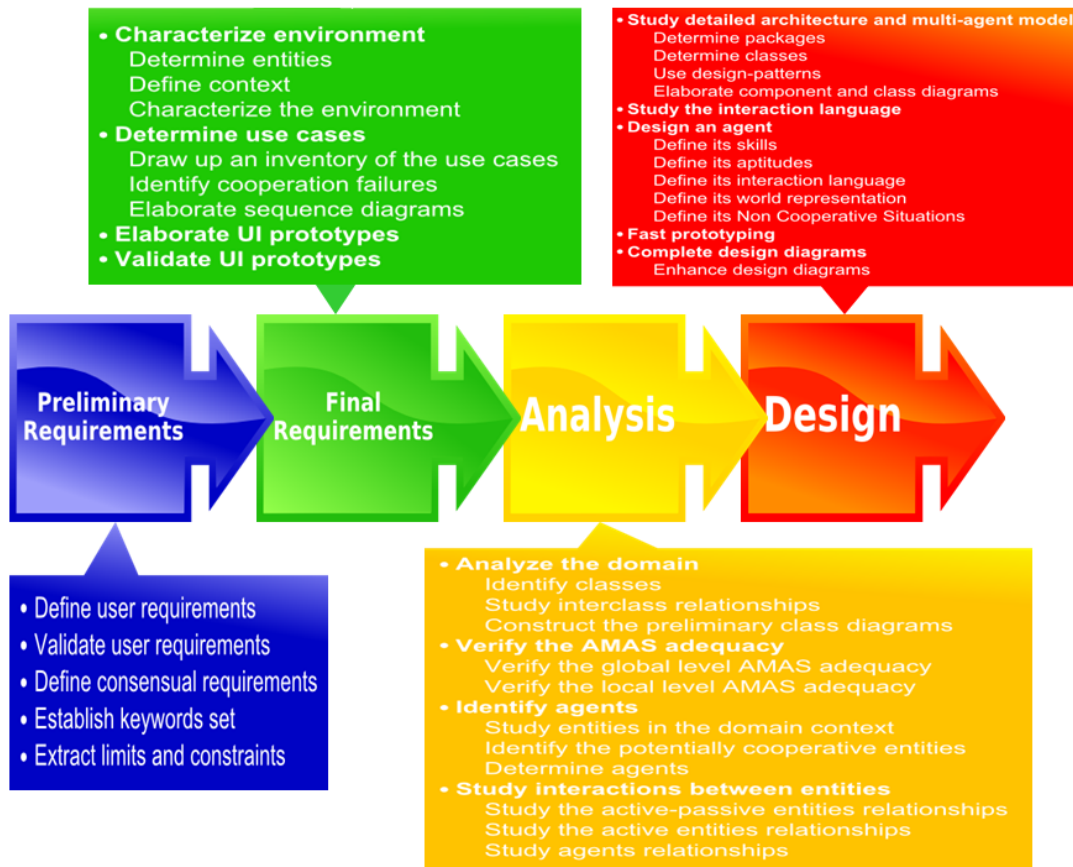


Figure 4.3: An overview of ADELFE [2]

Past the architectural design phase, the detailed design involves identifying: (*i*) an agent model, i.e. the set of agent classes in the MAS, implementing the identified roles, and the specific instances of these classes; and (*ii*) a services model, expressing services and interaction protocols to be provided within agent classes. The result of the design phase is assumed to be something that could be implemented in a technology neutral way.

4.2.2 ADELFE

ADELFE [9, 173] (see also Section 6.2) is based on object-oriented methodologies, follows the Rational Unified Process (RUP) [110] and uses UML [143] and AUML [64] notations. Some steps have been added in the classical workflows to be specific to adaptive MAS (Figure 4.3).

Adaptive software is used in situations in which the environment is unpredictable or the system is open. To solve these problems ADELFE guarantees that the software is developed according to the AMAS (Adaptive Multi-Agent System) theory. According to

this theory, building a system which is functionally adequate (which realises the right desired global function) is achieved by designing agents with a cooperation-driven social attitude. Agents composing an AMAS ignore the global function of the system, only pursue a local goal and try to always keep cooperative relations with one another. They are called *cooperative agents*. The cooperative agents are equipped with *five modules* to represent “physical”, “cognitive”, or “social” capabilities:

- *Skill Module*: represents knowledge on specific fields that enables agents to realise their partial function
- *Representation Module*: enables an agent to create its own representation about itself, other agents, or the environment it perceives
- *Interaction Module*: is composed of perceptions and actions.
- *Aptitude Module*: provides capabilities to reason about perceptions, skills, and representations – for example, to interpret messages
- *Cooperation Module*: embeds local rules to be locally *cooperative*. Cooperative means that agents are able to recognise “cooperation failures”

In ADELFE the requirements workflow is a fundamental step in software engineering. In the AMAS theory, the adaptation process starts from the interactions between the system and its environment. Therefore, it is important to give a model of the environment during this workflow. The environment model consists in three steps: determining the actors, defining the context and characterising the environment. In the analysis workflow, two steps are added to the RUP: the identification of the agents and the adequacy of the AMAS theory. ADELFE focuses on the identification of the agents. In the previous workflow, the designer has identified the entities of the system, now ADELFE must help him to identify what entities will be agents. In ADELFE, the notion of agent is restrictive: cooperative agents are the only kinds of agents considered. The designer has some guidelines: an agent is an entity previously defined, and this entity may be faced with unexpected events and it may have evolutionary representations about itself, other agents or about its environment and/or evolutionary skills. Because an adaptive MAS is not a technical solution for every application, ADELFE is the only methodology providing a tool to help a designer to decide if the AMAS theory is adequate to implement his application. For example, if the algorithm required to solve the task is already known, if the task is not complex, if the system is closed or if no unexpected events can occur, this kind of programming is useless.

In the design workflow, the agent model and the Non Cooperative Situations model are added to the RUP. ADELFE is dedicated to a specific architecture of agent: cooperative ones. Using the agent model, the designer must then describe the architecture of a cooperative agent by giving the components realising its behaviour. A MAS which is

not in a cooperative interaction with its environment needs to adapt itself to it. But, according to the AMAS theory, the global function of the system is not coded, only the local behaviour of the agents composing the society is coded. The adaptation will then be managed by these agents through the “Non Cooperative Situations” (NCS) model. An agent which locally detects cooperative failures acts to change its interaction with others to remove this state. The NCS of an agent must be described by the designer since they depend on the application. To help him, ADELFE provides the designer with generic cooperative failures such as incomprehension, ambiguity, uselessness or conflict. ADELFE also provides tables with fields to fill up concerning the name of the NCS, its generic type, the state in which the agent must be to detect it, the conditions of its detection and what actions the agent must perform to treat it.

Finally implementation and test workflows are similar to what is done in the RUP. Systems are realised through implementation of components. The process describes how you reuse existing components, or implement new components with well defined responsibility, making the system easier to maintain, and increasing the possibilities to reuse.

The tests are done throughout the project. This allows to find defects as early as possible, which radically reduces the cost of fixing the defect. Tests are carried out along four quality dimensions: reliability, functionality, application performance, and system performance. For each of these quality dimensions, the process describes how to go through the test lifecycle of planning, design, implementation, execution and evaluation.

4.2.3 Tropos

The Tropos methodology [13, 77] is intended to support all analysis and design activities in the software development process, from application domain analysis down to the system implementation (see also Section 6.3). In particular, Tropos rests on the idea of building a model of the system-to-be and its environment, that is incrementally refined and extended, providing a common interface to various software development activities, as well as a basis for documentation and evolution of the software. Tropos is composed by five main development phases: Early Requirements, Late Requirements, Architectural Design, Detailed Design and Implementation (Figure 4.4).

Requirements analysis represents the initial phase in many software engineering methodologies. As with other approaches, the ultimate objective of requirements analysis in Tropos is to provide a set of functional and non-functional requirements for the system-to-be.

Requirements analysis in Tropos is split in two main phases: Early Requirements and Late Requirements analysis. Both share the same conceptual and methodological approach. Thus most of the ideas introduced for early requirements analysis are used for late requirements as well. More precisely, during the first phase, the requirements engineer identifies the domain stakeholders and models them as social actors, who depend on one another for goals to be achieved, plans to be performed, and resources to be furnished. By clearly defining these dependencies, it is then possible to state the why, beside the

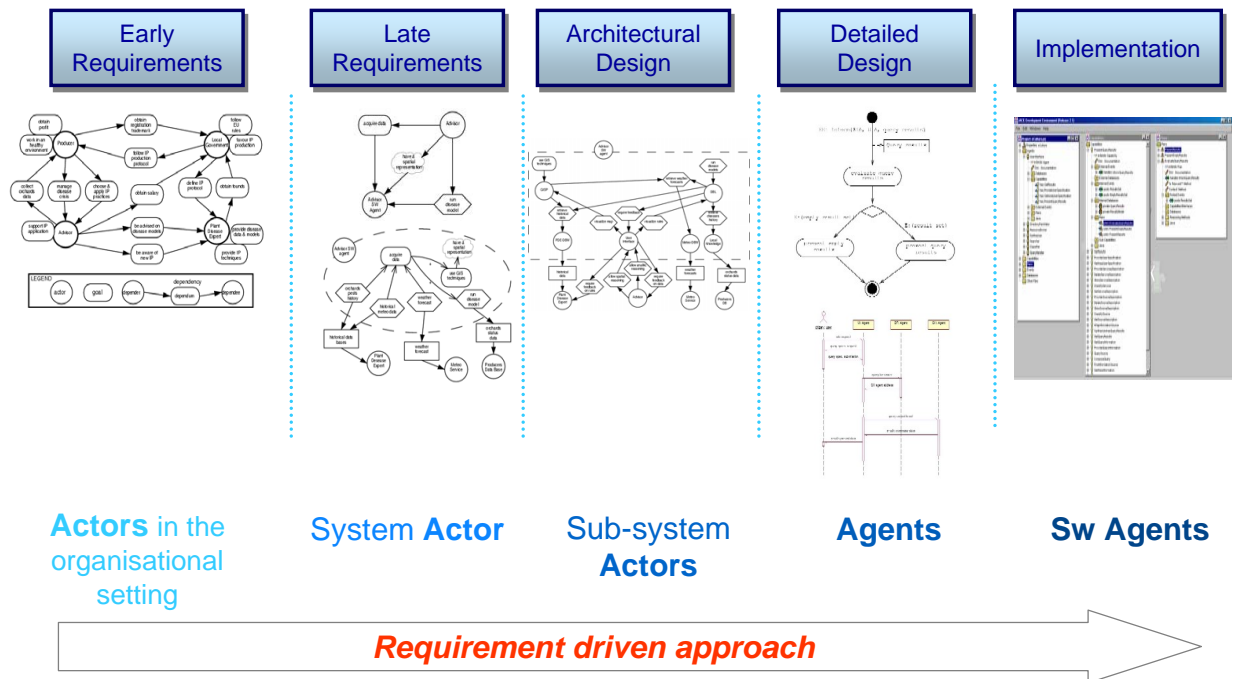


Figure 4.4: An overview of Tropos [211]

what and how, of the system functionalities and, as a last result, to verify how the final implementation matches initial needs. In the Late Requirements analysis, the conceptual model is extended including a new actor, which represents the system, and a number of dependencies with other actors of the environment. These dependencies define all the functional and non-functional requirements of the system-to-be.

The Architectural Design and the Detailed Design phases focus on the system specification, according to the requirements resulting from the above phases. Architectural Design defines the system's global architecture in terms of sub-systems, interconnected through data and control flows. Sub-systems are represented, in the model, as actors and data/control interconnections are represented as dependencies. The architectural design provides also a mapping of the system actors to a set of software agents, each characterized by specific capabilities. The Detailed Design phase aims at specifying agent capabilities and interactions. At this point, usually, the implementation platform has already been chosen and this can be taken into account in order to perform a detailed design that will map directly to the code.

The Implementation activity follows step by step, in a natural way, the detailed design specification on the basis of the established mapping between the implementation platform constructs and the detailed design notions.

4.2.4 PASSI

PASSI (Process for Agent Societies Specification and Implementation) [35, 38] (see also Section 6.1) is a step-by-step methodology for the design and development of multi-agent societies (Figure 4.5).

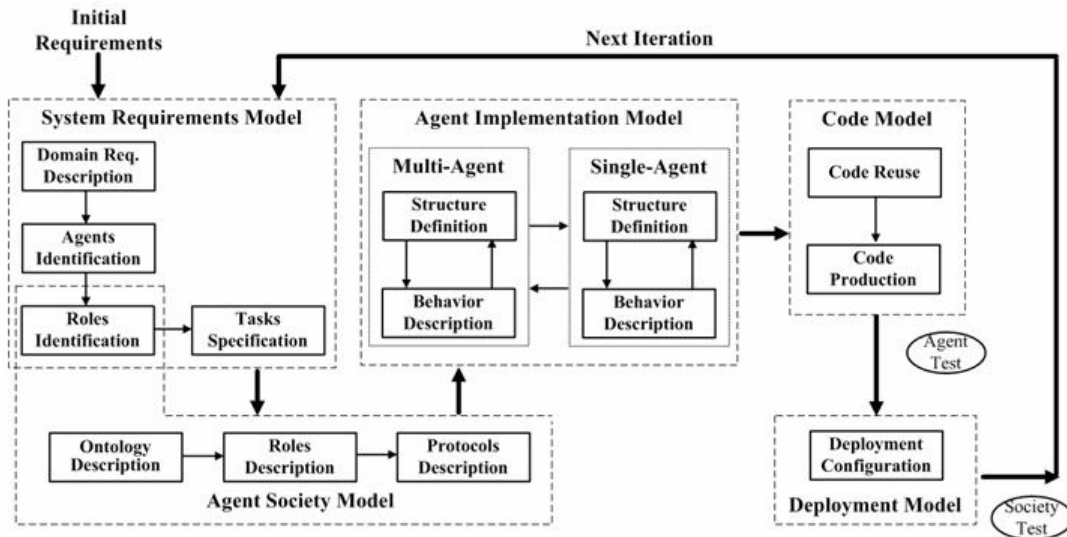


Figure 4.5: An overview of PASSI [168]

PASSI has adopted UML as the modelling language since it is widely accepted both in the academic and industrial environments. Its extension mechanisms (constraints, tagged values and stereotypes) helps in customizing the representations of agent-oriented designs so as to avoid the adoption of a totally new modelling language. The models of PASSI are: System Requirements Model, Agent Society Model, Agent Implementation Model, Code Model and Deployment Model.

The System Requirements Model is anthropomorphic of the system requirements in terms of agency and target. It comprises four steps:

- *Domain Requirements Description*: a functional description of the system made by using conventional use case diagrams
- *Agent Identification*: the phase of attribution of responsibilities to agents, represented as stereotyped UML packages
- *Role Identification*: a series of sequence diagrams exploring the responsibilities of each agent through role-specific scenarios
- *Task Specification*: specification of the capabilities of each agent with activity diagrams

The Agent Society Model is the model of the social interactions and dependencies between the agents playing a part in the solution. It necessitates three steps:

- *Ontology Description*: use of class diagrams and OCL constraints to describe the knowledge ascribed to individual agents and their communications
- *Role Description*: class diagrams are used to show the roles played by agent, the task involved, communication capabilities and inter-agent dependency
- *Protocol Description*: use of sequence diagrams to specify the grammar of each pragmatic communication protocol in terms of speech-act performative

The Agent Implementation Model is a model of the solution architecture in terms of classes and methods; the most important differences with common object-oriented approach is that there are two different levels of abstraction, the social (multi-agent) level and the single level. This model is composed by:

- *Agent Structure Definition*: conventional class diagrams describe the structure of solution agent classes
- *Agent Behaviour Description*: activity diagrams or state charts describe the behaviour of individual agent

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

- Generation of code from the model using one of the functionalities of the PASSI add-in
- It is possible to generate not only the skeletons but also largely reusable parts of the method's implementation based on a library of reused patterns and associated design description
- Manual completion of the source code

Finally the Deployment Model is a model of the distribution of the system's parts across hardware processing units, and of their migration. The only step in this model is the *Development configuration* in which the deployment diagrams describe the allocation of agents to the available processing units and any constraints on migration and mobility.

4.2.5 MaSE

The Multiagent System Engineering (MaSE) methodology [44, 135], takes an initial system specification, and produces a set of formal design documents in a graphically based style. An overview of the methodology and models is shown in Figure 4.6.

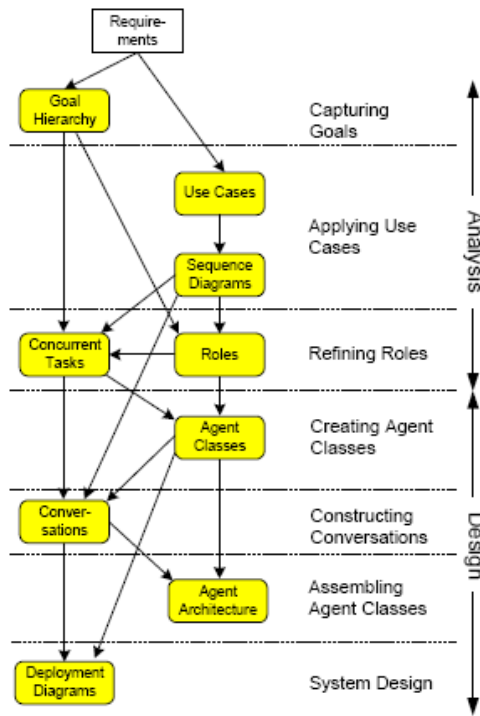


Figure 4.6: An overview of MaSE [135]

The first phase in MaSE is Capturing Goals, which takes the initial system specification and transforms it into a structured set of system goals as shown in a Goal Hierarchy Diagram. A goal is always defined as a system-level objective. Lower-level constructs may inherit or be responsible for goals, but goals always have a system-level context. There are two parts of the Capturing Goals phase: identifying and structuring goals. The goals are identified by distilling the essence of the set of requirements. Once these goals have been captured and explicitly stated, they are less likely to change than the detailed steps and activities involved in accomplishing them.

The second phase of MaSE looks down the road toward constructing conversations between agents that are the real backbone of a MAS, and creates use cases to ease this difficulty. The Applying Use Cases phase captures use cases from the initial system requirements and restructures them as a Sequence Diagram. A sequence diagram depicts a sequence of messages between multiple agent roles.

The third step of MaSE is to transform the structured goals of the Goal Hierarchy Diagram into a form more useful for constructing MAS: roles. Roles are the building blocks used to define agent classes and capture system goals during the design phase. The general case transformation of goals to roles is one-to-one; each goal maps to a role. However, there are many exceptional situations where it is useful to combine goals. Similar or related goals may be combined into single roles for the sake of convenience or efficiency.

Role definitions are captured in a traditional Role Model. The Role Model is useful at the outset of the role definition process before tasks have been defined, as well as later in the analysis to provide a high-level view of the system. After roles are created, tasks are associated with each role. Every goal associated with a role can have a task that details how the goal is accomplished. A task is a structured set of communications and activities, depicted as a state diagram. Subsequently in the Creating Agent Classes phase, the agent classes are identified from component roles. The product of this phase is an Agent Class Diagram which depicts agent classes and the conversations between them. During this phase of MaSE, agent classes consist of two components: roles and conversations.

Constructing Conversations is the next phase of MaSE. It is closely linked with the phase that follows it, Assembling Agents. A MaSE conversation defines a coordination protocol between two agents. Specifically, a conversation consists of two Communication Class Diagrams, one each for the initiator and responder. A Communication Class Diagram is a pair of finite state machines that define the conversation states of the two participant agent classes. After that, in the Assembly Agent phase of MaSE, the internals of agent classes are created. MaSE defines five different architectural style templates: Belief-Desire-Intention (BDI), reactive, planning, knowledge based, and a user-defined architecture. Each architecture template has a specific set of components. As discussed above, constructing conversations and agent class assembly are closely related activities. In practice, it is useful to alternate between these phases while staying within one functional area of the design. The designer should design conversations first if the system consists of many simple conversations, or if the initial context of the system includes many use cases. It is generally better to define the agents first if there are complex conversations, or if many of the agent classes are being reused.

The final phase of the MaSE methodology takes the agent classes and instantiates them as actual agents. It uses a Deployment Diagram to show the numbers, types, and locations of agents within a system. System design is actually the simplest phase of MaSE, as most of the work was done in previous steps. The idea of instantiating agents from agent classes is the same as instantiating objects from object classes in object-oriented programming. Deployment Diagrams are used to define a system based on agent classes defined in the previous phases of MaSE. Deployment Diagrams define system parameters such as the actual number, types, and locations of the agents within the system.

4.2.6 INGENIAS

A MAS in INGENIAS [169, 170] is considered from five viewpoints (see Figure 4.7): organisation, agent, goals/tasks, interactions, and environment. For each viewpoint, a set of concepts and relationships among them are provided to the developer to describe the MAS; the agent-oriented software developer can use those concepts and others such as agent, organisation, goal, task, mental state, resource, etc. These concepts may appear in the specification of one or several viewpoints, therefore the need to identify consistency

rules among these.

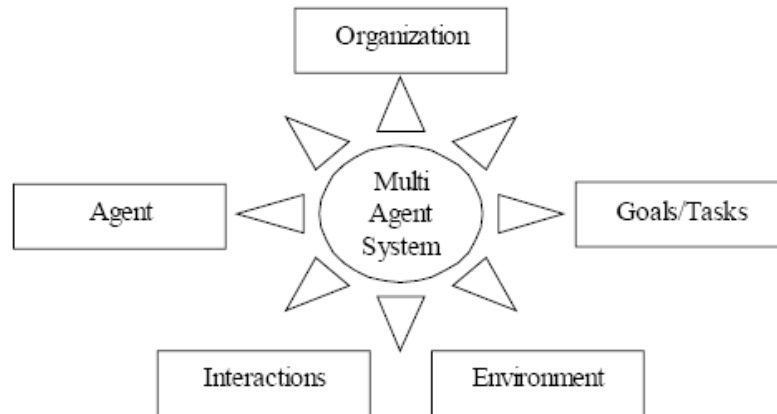


Figure 4.7: An overview of Ingenias [81]

The organization describes a structure for agents, resources, goals and tasks. It is defined by:

- Purpose and tasks of the organisation. An organisation has one or several purposes, a goal, and is capable of performing certain tasks to achieve them
- A structure of groups and workflows. From a structural viewpoint, the organisation is a set of entities with relationships of aggregation and inheritance. This structure defines a framework where agents, resources, goals, and tasks are placed. And on this structure there are relationships that induce the formation of workflows and social rules
- Workflows define associations among tasks and general information about their execution
- Groups may contain agents, roles, resources, or applications.
- Social relationships. They can be established at different levels: between organisations, groups, agents, or roles. There are service relationships, conditional or unconditional subordination, etc. They state restrictions on the interactions between entities.

The agent viewpoint is concerned with the functionality of each agent: responsibilities (what goals an agent is forced to pursue) and capabilities (what tasks it is able to perform). The behaviour of the agent is defined through three components:

- Mental state, an aggregation of mental entities such as goals, beliefs, facts, compromises
- Mental state manager, which provides for operations to create, destroy and modify mental entities
- Mental state processor, which determines how the mental state evolves, and can be described in terms of rules, planning, etc.

Mental state can be seen as all the information that allows the agent to take decisions. This information is managed and processed in order to produce agent decisions and actions, by the mental state manager and processor.

The tasks/goals viewpoint considers the decomposition of goals and tasks, and describes the consequences of performing a task, and why it should be performed (i.e., it justifies the execution of tasks as a way to satisfy goals, and these change as tasks execute). The mental state processor takes the decision of which task to execute, and the mental state manager provides the operations to create, destroy and modify the elements of the mental state and their relationships. In fact, a goal may have associated a life-cycle.

The interaction viewpoint addresses the exchange of information or requests between agents, or between agents and human users. The definition of an interaction requires the identification of:

- Actors in the interaction: who plays the role of initiator and who are the collaborators. Which is the reason (goal) that motivates each actor to participate in the interaction
- Definition of interaction units: messages, speech acts
- Definition of the possible orders of interactions units: protocols
- Which actions are performed in the interaction
- Context of the interaction: which goal pursues the interaction and which are the mental states of its participants
- Control model: coordination mechanisms

Finally, the environment viewpoint defines the entities with which the MAS interacts, which can be:

- Resources. Elements required by tasks that do not provide a concrete API. Examples of resources are the CPU, File descriptors, or memory. Resources are assigned to agents or groups in the current system.

- Applications. Normally they offer some (local or remote) API. In some cases, applications are wrapped by agents to facilitate their integration in the MAS. Their main use is to express the perception of the agents: applications produce events that can be observed. Agents define their perception indicating which events they listen to
- Other agents (from other organisations). The agents in the MAS interact with these agents to satisfy system functionality

4.2.7 MESSAGE

The MESSAGE methodology [19, 69] has adopted the Rational Unified Process (RUP) 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 (Figure 4.8).

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 organisations 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 behaviour. 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.

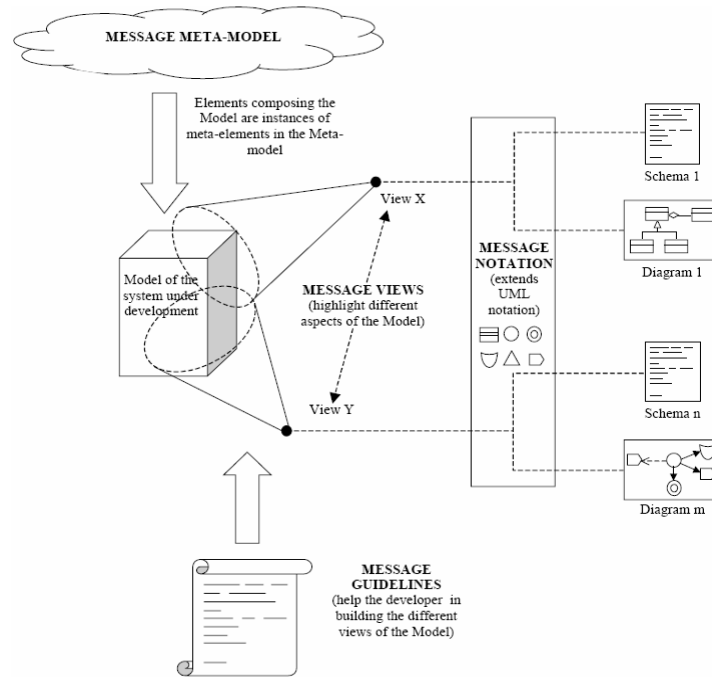


Figure 4.8: An overview of MESSAGE [19]

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.

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 organisations that interact with resources, actors, or other organisations. Actors may be human users or other existing agents. Subsequent stages of refinement result in the creation of models at level 1, level 2 and so on [19].

A MESSAGE analysis model is a complex network of inter-related classes and instances. MESSAGE defines a number of views that focus on overlapping sub-sets of entity and relationship concepts (Figure 4.8). Five views have been defined to help the modeler focus on coherent subsets of the modelling language: Organisation, Goal/Task, Agent/Role, Interaction, and Domain.

The Organisation view shows concrete entities, i.e., Agents, Organisations, 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 behaviour 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: Organisation, Goal/Task, Delegation and Interaction, which extend the UML Class diagram, and Workflow which extends the UML Activity diagram.

The MESSAGE Design Model redefines 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 organisation. 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 behaviour. 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.

4.2.8 Prometheus

The Prometheus methodology [163, 164] is a detailed process for specifying, designing, and implementing intelligent agent systems. The goal in developing Prometheus is to have a process with defined deliverables which can be taught to industry practitioners and undergraduate students who do not have a background in agents and which they can use to develop intelligent agent systems.

Prometheus supports the development of intelligent agents, providing “start-to-end” support, having evolved out of practical industrial and pedagogical experience, having been used in both industry and academia, and, above all, in being detailed and complete. Prometheus is also amenable to tool support and provides scope for cross checking between

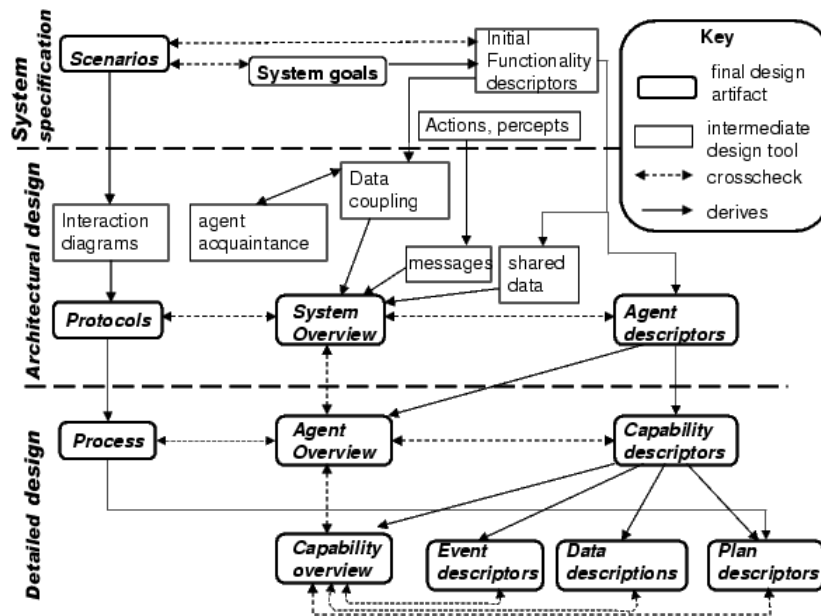


Figure 4.9: An overview of Prometheus [49]

designs, and uses only UML diagrams without any support for standard languages.

The methodology consists of three phases (Figure 4.9):

- *System Specification* phase, which involves several activities:
 - Identify system goals and sub-goals
 - Develop use case scenarios
 - Identify the agent system’s interface to the environment in terms of actions, percepts, and external data
 - Identify functionalities
 - Identify data read and written by functionalities
 - Prepare functionality schemas (name, description, actions, percepts, data used/produced, interaction, and goals)
- *Architectural Design* phase, which involves several activities:
 - Group functionalities to determine agent types using data coupling and agent acquaintance diagrams to assess alternative groupings
 - Define agent types (also define the number and life-cycle of the agent types) and develop agent descriptors

- Produce a system level overview diagram describing the overall structure of the system
- Develop interaction protocols from use case scenarios (via interaction diagrams)
- *Detailed Design* phase, which involves several activities:
 - Develop process diagrams
 - Produce agent overview diagrams showing the internal workings of agents in terms of capabilities, events, data and plans
 - Refine capability internals (add included capabilities and interactions)
 - Introduce plans to handle events
 - Define details of events (external, between agents, between capabilities and within agents)
 - Define details of plans (relevance, context, subgoals)
 - Define details of beliefs/data

In Prometheus an aspect of system specification is the use case scenarios, which describe examples of the system in operation and are a variant of the scenario part of UML's use cases [163]. In the Architectural Design phase the interaction between agents are defined using interaction diagrams and interaction protocols. The notation used to do this is a simplified variant of UML sequence diagrams for interaction diagrams, and AUML for the interaction protocol. Prometheus describes structural overviews at various levels (system, agent, capability) with a single diagram type. In addition, diagrams are used to show data coupling and agent acquaintance relationship. Dynamic behaviour is described with existing models from UML and AUML.

System's goals and functionalities are determined in the System Specification phase, and they are strongly considered. The determination of goal is an iterative process: identify and refine system goals, group goals into functionalities, prepare functionality descriptor, define use case scenarios (that help to identify missing goals), check that all goals are covered by scenarios. An initial set of goals is identified from the initial requirements; there are refined and elaborated into hierarchy of goals by asking how goals will be achieved, and why goals are being achieved.

Here roles are defined as functionalities, analysed during the System Specification phase. They have to be kept as narrow as possible, dealing with a single aspect or sub-goal of the system, in fact, if functionalities are too broad they are likely to be less adequately specified leading to potential misunderstanding. In defining a functionality, it is important to define information required and produced by it; each functionality should be linked to some system goals [163]. Roles definition is also used in the Architectural Design phase, in constructing a Data Coupling Diagram that consists of the functionalities and all identified data, but they are not so deeply treated.

4.3 Methodologies Comparison

In the last years several methodologies evaluations are proposed in the literature. Tran and Low [140, 210] have proposed an evaluation framework based on process-related criteria, technique-related criteria, model-related criteria and supportive-feature criteria.

Sturm and Shehory [202, 203] have proposed an evaluation based on four major division: concepts and properties, notations and modelling techniques, process, and pragmatics. The authors have highlighted that the evaluation of methodologies introduces several difficulties: i) methodologies might address different aspects or differ in their terminology, ii) some of the methodologies are influenced by a specific approach, e.g., BDI or OO, iii) the completeness of various methodologies varies dramatically. So, comparing methodologies is a very complex tasks.

Here we adopt the following criteria for evaluating the methodologies presented in this chapter:

- Lifecycle criteria (Subsection 4.3.1).
- Notation criteria (Subsection 4.3.2).

A summary of this section is reported in Subsection 4.3.3

4.3.1 Lifecycle Criteria

These criteria look at the development approach followed by the methodologies and the covered steps. These criteria are important because specify the support provided by the methodology during the software lifecycle and evaluate the kind of process supported by the methodology. In particular:

- *Development Lifecycle* expresses what development lifecycle best describes the methodology (e.g., iterative or sequential).
- *Coverage Lifecycle* specifies what phases of the lifecycle are covered by the methodology.
- *Development Perspective* specifies what development perspective is supported by the methodology (e.g., top-down, bottom-up or hybrid).
- *Support for verification* specifies if the development process of the methodology contains rules to allow for the verification of the correctness of the developed models and specifications.

Figure 4.10 summarises the results of the comparative analysis of the methodologies presented in this chapter. ADELFE, MESSAGE and INGENIAS adopt a formally defined, prominent development lifecycle (RUP or the Unified Software Development Process

	Development Lifecycle	Coverage Lifecycle	Development Perspective	Support for Verification
Gaia	sequential	Analysis Design	top-down	No
Tropos	iterative	Analysis Design Implementation	top-down	Yes
Prometheus	iterative	Analysis Design	bottom-up	Yes
PASSI	iterative	Analysis Design Implementation	bottom-up	Yes
ADELFE	RUP	Analysis Design Implementation	top-down	Yes
MaSE	iterative	Analysis Design	top-down	Yes
MESSAGE	RUP	Analysis Design Implementation	hybrid	Mentioned as future enhancement
INGENIAS	USDP	Analysis Design Implementation	hybrid	Yes

Figure 4.10: Comparison regarding lifecycle criteria

(USDP)), the other methodologies describe informally their lifecycle. In both cases the development process of all methodologies involves a high degree of iteration within and/or across the development phases.

Tropos, PASSI, ADELFE, MESSAGE and INGENIAS cover all the software lifecycle, while the other methodologies provide support only for the analysis and the design phases. A designer should consider the lifecycle coverage when choose a methodologies, because the phases not covered by methodologies should be managed in an ad way by the designer.

The development perspective is different in the considered methodologies: Gaia, Tropos, ADELFE and MaSE follow a top-down approach – an overview of the system is first formulated, specifying but not detailing any first-level subsystems, each subsystem is then refined in yet greater detail – while Prometheus and PASSI follow a bottom-up approach—the individual base elements of the system are first specified in great detail, then linked together to form larger subsystems, which then in turn are linked, sometimes in many levels, until a complete top-level system is formed. Indeed, MESSAGE and INGENIAS adopts an hybrid approach providing support for different system’s views.

	Notation	Easy to understand	Usability	Supporting Tool
Gaia	ad hoc	high	medium	No
Tropos	ad hoc	medium	medium	Yes
Prometheus	UML, AUML	high	high	Yes
PASSI	UML	high	high	Yes
ADELFE	UML, AUML	high	high	Yes
MaSE	UML, AUML	high	medium	No
MESSAGE	ad hoc	medium	medium	Yes
INGENIAS	ad hoc	medium	medium	Yes

Figure 4.11: Comparison regarding notation criteria

Finally, only Gaia and MESSAGE do not provide support for the verification of the correctness of the developed models and specifications.

4.3.2 Notation Criteria

These criteria look at the notations adopted by methodologies, supporting tools and usability (Figure 4.11). These criteria are important because sometimes the notation adopted could establish the methodology’s success. In particular:

- *Notation* specifies the type of notation adopted by the methodology.
- *Easy to understand* specifies if the notation adopted is easy to understand for the new users.
- *Usability* specifies the degree of usability of the methodology.
- *Supporting tool* specifies if exist one ore more tools that support the methodology.

Figure 4.11 summarises the results about the notation criteria of the methodologies presented in this chapter.

Gaia, Tropos, MESSAGE and INGENIAS adopt an ad hoc notation for expressing the outcomes of the development steps, while the other methodologies use a standard notation like UML and AUML (Agent-UML). The kind of notation influences the “easy to understand” criterion, in fact the methodologies that adopt a standard notation are easier to understand than the others—except for Gaia, that has an high level of understanding. This because for new users that typically come from the object-oriented field it is easier to understand the standard UML or its variant for agent than to learn a new notation. Obviously also the usability of the methodology is influenced by the notation adopted. In fact, the methodologies that adopt a standard notation are more usable than the other because the startup time – composed by the time to learn methodology plus the time

to learn its notation – is typically higher in those methodologies that adopts an ad hoc notation where the time for learning a new notation becomes relevant.

Finally, only Gaia and MaSE have not a supporting toolkit. This could be a limitation because a methodology without a supporting tool is less usable than a methodology supported by a tool.

4.3.3 Summing up

This section has presented a comparison of the AO methodologies presented in Section 4.2. As for software development, individual methodologies are often created with specific purposes in mind [85]: particular domains or particular segments of the lifecycle. However users often make the assumption that a methodology is not in fact constrained but, rather, is universally applicable. This can easily lead to methodology failure, and to the partial rejection of methodological thinking by software development organisation. As highlighted in Subsection 3.2.2, the creation of a single universally applicable methodology is an unattainable goal. The question is how could we create a methodological environment in which the various demands of different software developers might be satisfied altogether. So, the decision about the best methodology should depend on the target application. Each application entails a different set of requirements that indicate which evaluation criteria are the most important and should be supported by the chosen methodology.

As a final remark, the comparison presented here has the following limitations:

- the comparison is solely based upon the available documentations of the methodologies and their documented case studies;
- some evaluation criteria are subjective in nature, particularly the usability and understandability of the methodologies;
- the criteria adopted do not cover all the possible methodologies' key features, but only those considered relevant in the context of this chapter. In fact two other comparisons are provided in Chapters 8 and 12 respectively about how methodologies supports the design of the environment and manege the complexity of the representation.

Part II

Meta-models

5

Meta-models & Languages

This chapter introduces the concept of meta-model and the different types of languages for describing meta-models. In particular Section 5.1 presents several definition of the meta-model, the level of abstraction inducted by meta-models, and illustrates the key role that meta-models play in the context of methodologies and their evaluations and integrations. Section 5.2 presents different types of meta-modelling languages. Like other modelling languages, meta-modelling languages focus on specific aspects of the domain to be modelled, and therefore lead to different types of representations. For example the meta-models that represent the abstractions adopted by methodologies are different from those meta-models that represent the software development processes. This means that different kinds of meta-modelling languages are necessary in order to capture the distinctiveness of each domain that requests a meta-model.

The meta-modelling technique could be very useful also for representing the abstractions supported by infrastructures in order to both study the deep infrastructure semantics and evaluate the gap between methodologies and infrastructures. There is the need to study the different AO infrastructures, to compare their abstractions, rules, relationships, and the process they follow, and to have a comprehensive view of this variety of infrastructures (Section 10.5). So, Section 5.3 presents how to meta-modelling infrastructures.

The use of a meta-model formalism helps to compactly and precisely express each methodology and infrastructure and provides a basis for analysing and comparing them. It also helps to study the existing gap between AO methodologies in general and AO infrastructures, and provides a starting point for developing methodologies together with their counterpart AO infrastructures.

Finally a summary follows in Section 5.4.

5.1 Meta-Models

Software development processes and methodologies have always been described in terms suitable for use by the developer [88]. They talk about what tasks and techniques should be used, what sort of lifecycle is appropriate (e.g. waterfall) and how these process elements should be organised in time and assigned to people. They are often described in

a manual or published as a book that the project manager and his/ her team of developers follow closely. Previous comparisons of OO processes for software development have been undertaken, for example, by Henderson-Sellers et al. [88] and by Hull et al. [93].

With the advent of CASE tools it became necessary to create a rule base within each tool that would support these processes. These rules would say whether it was appropriate to, for example, sequence two activities, three techniques and then four roles (an occurrence that should be prevented since it is nonsensical). These rules are currently commonly captured in a meta-model. In its most general acceptance, a meta-model is defined as

Wikipedia (1) Meta-modeling is the analysis, construction and development of the frames, rules, constraints, models and theories applicable and useful for the modeling in a predefined class of problems.

(2) meta-modelling is the construction of a collection of *concepts* (things, terms, etc. . .) within a certain domain. A model is an abstraction of phenomena in the real world, and a metamodel is yet another abstraction, highlighting properties of the model itself. This model is said to conform to its metamodel like a program conforms to the grammar of the programming language in which it is written. Common uses for meta-models are:

- As a schema for semantic data that needs to be exchanged or stored,
- As a language that supports a particular method or process,
- As a language to express additional semantics of existing information.

Metamodels.com A meta-model is a precise definition of the constructs and rules needed for creating semantic models.

Bernon et al. The process of designing a system (object or agent-oriented) consists of instantiating the system meta-model that the designers have in their mind in order to fulfill the specific problem requirements. In the agent world this means that the meta-model is the critical element because of the variety of methodology meta-models [10].

Gonzalez-Perez et.al A meta-model is a model of a methodology or, indeed, of a family of related methodologies [80].

Brian Henderson-Sellers A meta-model describes the rules and constraints of meta-types and meta-relationships. Concrete meta-types are instantiated for use in regular modelling work. A meta-model is at a higher level of abstraction than a model. It is often called *a model of a model*. It provides the rules/ grammar for the modelling language itself. The modelling language consists of instances of concepts in the meta-model.

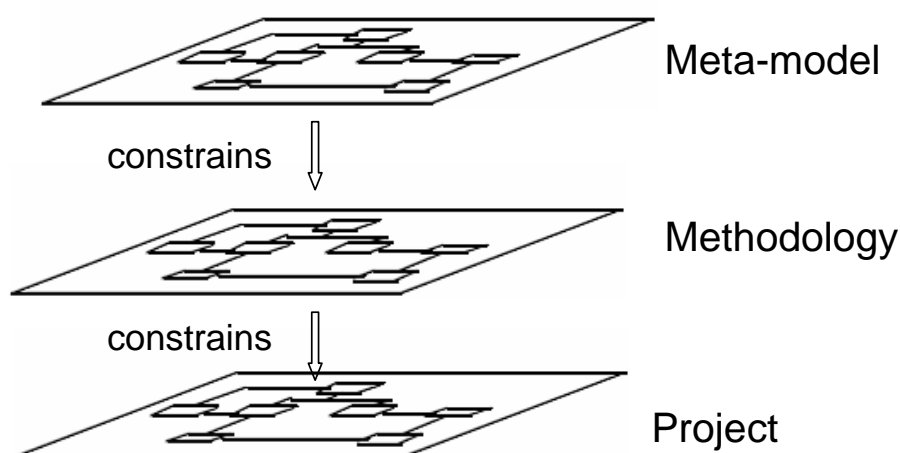


Figure 5.1: Level of abstractions in the Meta-models

It's also important to understand that meta-models are always made for a particular purpose. Do not ever attempt to use a meta-model without understanding the particular goal that the authors had in mind when they created the meta-model.

Although it is possible to describe a methodology without an explicit meta-model, formalising the underpinning ideas of the methodology in question is valuable when checking its consistency or when planning extensions or modifications. A good meta-model should address all of the different aspects of methodologies, i.e. the process to follow and the work products to be generated. In turn, specifying the work products that must be developed implies defining the basic modelling building blocks from which they are built. The importance of meta-model becomes clear when it is necessary to study the completeness and the expressiveness of a methodology, and when comparing different methodologies. Meta-models are extremely important for integrating methodologies, too. Integrating methodologies without such a formalism might lead to two kinds of errors: assuming an existence of differences of concerns when none exists, and the wrong assumption of concerns similarity. The first type of errors might lead to repetition and consequently an unnecessarily large methodology that is hardly understood and acceptable by developers. The second type of errors will lead to inconsistency because of the false assumption of concerns interpretation similarity.

Meta-models are often used by methodologists to construct or modify methodologies. In turn, methodologies are used by software development teams to construct software products in the context of software projects. Meta-model, methodology and project constitute, in this approach, three different areas of expertise that, at the same time, correspond to three different levels of abstraction and three different sets of fundamental concepts (Figure 5.1) [88]. As the work performed by the development team at the project level is constrained and directed by the methodology in use, the work performed

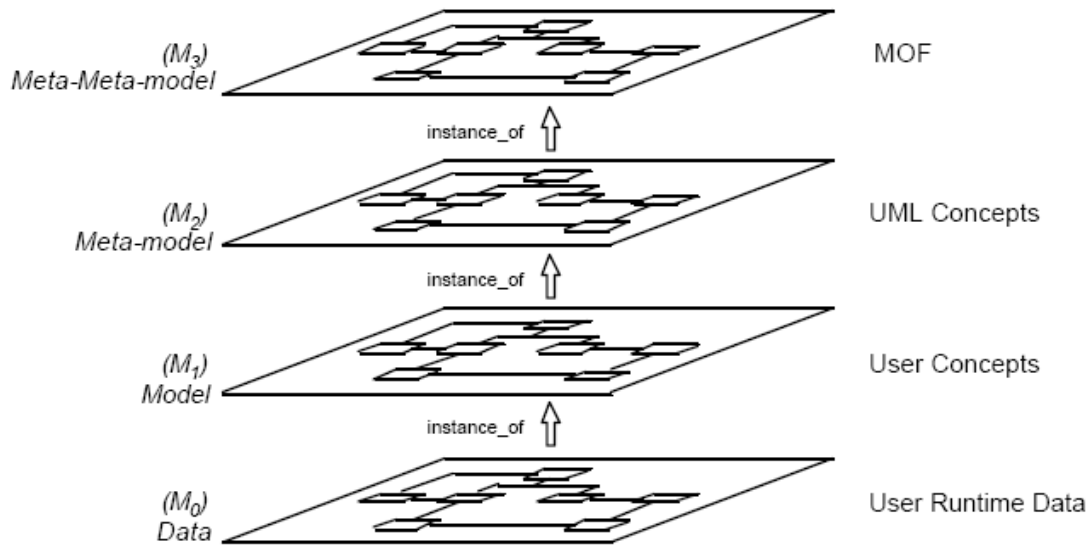


Figure 5.2: The OMG's layers

by the methodologist at the methodology level is constrained and directed by the chosen meta-model (Figure 5.1).

Traditionally, these relationships between *modelling layers* are seen as *instance-of* relationships, in which elements in one layer are instances of some elements in the layer above. Most object-oriented process meta-modelling approaches define a metamodel as a model of a methodology that a software development team may employ. Following this conventional approach, *classes* in the meta-model are used by the methodologist to create instances in the methodology layer and thus generate a methodology. However, these objects in the methodology layer are often used as classes by the development team to create elements in the project layer during methodology enactment. An example is the Meta Object Facility an OMG standard (Figure 5.2) [142]. MOF is designed as a four-layered architecture. It provides a meta-meta model at the top layer, called the M3 layer. This M3-model is the language used by MOF to build meta-models, called M2-models. The most prominent example of a Layer 2 MOF model is the UML [143] meta-model, the model that describes the UML itself. These M2-models describe elements of the M1-layer, and thus M1-models. These would be, for example, models written in UML. The last layer is the M0-layer or data layer. It is used to describe the real-world. MOF is a closed meta-modeling architecture; it defines an M3-model, which conforms to itself. MOF allows a strict meta-modelling architecture; every model element on every layer is strictly in correspondence with a model element of the layer above. MOF only provides a means to define the structure of a language or of data.

5.2 Meta-Modelling Languages

This section presents different types of meta-modelling languages: Subsection 5.2.1 shows meta-modelling languages for modelling the methodologies abstractions and their relationships, while Subsection 5.2.2 depicts the languages for modelling the software development processes.

5.2.1 Meta-Modelling Languages for Abstractions

Several research efforts are being devoted to developing meta-models for the abstractions exploited by AO methodologies, however standardisations of methodologies for development of meta-models are not going still a long way off. Although UML is often used for that purpose, meta-modelling methodologies (and in particular agent-oriented methodologies) present several peculiarities.

A comparison among the meta-modelling power of UML and OPM is shown in Chapter 13, where the meta-models of the **SODA** in UML and in OPM methodology are depicted.

UML

The Unified Modelling Language (UML) [143] is a graphical language for visualising, specifying, constructing, and documenting the artifacts of a software-intensive system. The UML offers a standard way to write a system's blueprints, including conceptual things such as business processes and system functions as well as concrete things such as programming language statements, database schemas, and reusable software components.

The important point to note here is that UML is a *language* for specifying and not a methodology or procedure. The UML may be used in a variety of ways to support a software development methodology (such as the Rational Unified Process [110])—but in itself it does not specify that methodology or process.

UML diagrams represent three different views of a system model:

- Functional requirements view: emphasises the functional requirements of the system from the user's point of view. Includes use case diagrams.
- Static structural view: emphasises the static structure of the system using objects, attributes, operations, and relationships. Includes class diagrams and composite structure diagrams.
- Dynamic behaviour view: emphasises the dynamic behaviour of the system by showing collaborations among objects and changes to the internal states of objects. Includes sequence diagrams, activity diagrams and state machine diagrams.

In UML 2.0 there are thirteen types of diagrams. To understand them, it can be useful to categorise them hierarchically, as shown in Figure 5.3.

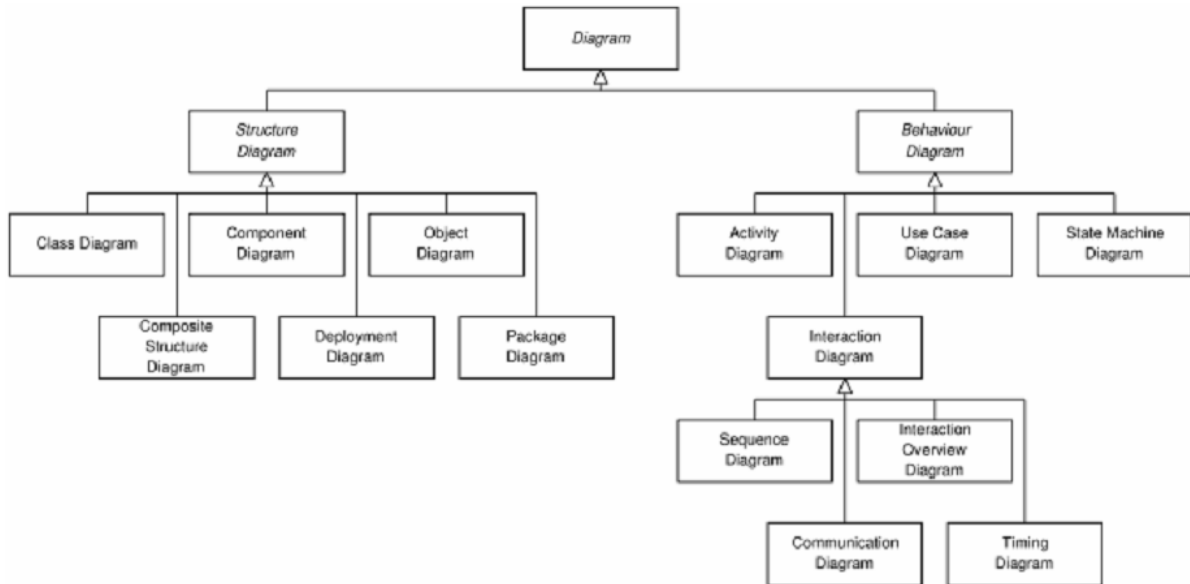


Figure 5.3: UML's diagrams

Structure diagrams emphasise what things must be in the system being modelled: Class diagram, Component diagram, Composite structure diagram, Deployment diagram, Object diagram, Package diagram. Behaviour diagrams emphasise what must happen in the system being modelled: Activity diagram, State Machine diagram, Use case diagram. Finally Interaction diagrams, a subset of behaviour diagrams, emphasise the flow of control and data among the things in the system being modelled: Communication diagram, Interaction overview diagram, Sequence diagram, UML Timing Diagram. The Protocol State Machine is a sub-variant of the State Machine. It may be used to model network communication protocols. UML does not restrict UML element types to a certain diagram type. In general, every UML element may appear on almost all types of diagrams. This flexibility has been partially restricted in UML 2.0.

The general adoption of UML as a world standard for system modelling makes it the first natural choice for representing meta-models.

Some specific issues are raised by adopting UML to express *meta-models of methodologies*, since representing a methodology is inherently different from representing a system at the object level. In particular, when meta-modelling methodologies, UML leads to emphasise objects and object relations, leaving aside the procedural aspects, which can be revealed only indirectly, by object operations and message exchanges. Moreover, the five behavioural diagrams provided by UML to capture the dynamic behaviour of a system at the object level become of little use at the meta-level, as they were defined to express which and how interaction occurs, rather than what interaction is and what role it plays—which is what is needed when representing a methodology. So, UML-based meta-models

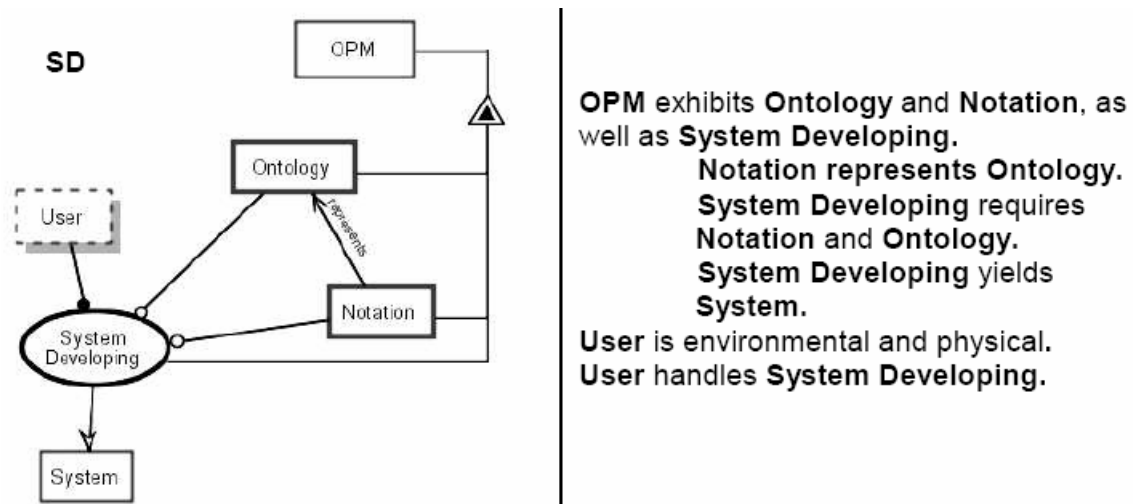


Figure 5.4: Top level specification of the OPM meta-model: the OPD (left) and the OPL(right) [54]

usually exploit only package diagrams, class diagrams, and associations.

OPM

The *Object Process Methodology* (OPM henceforth) [53] is an integrated approach to the study and development of systems in general, and of software systems in particular. OPM is also a reflective methodology, i.e. a methodology that can model itself without requiring any auxiliary means or external tools. OPM unifies the system's life-cycle stages (specification, design and implementation) within one single frame of reference, using a single diagramming tool – Object-Process Diagrams (OPDs) – and a corresponding textual language, called Object-Process Language (OPL) (Figure 5.4). A set of inter-related OPDs, constitute the graphical, visual OPM formalism. Each OPM element is denoted in an OPD by a dedicated symbol, and the OPD syntax specifies correct and consistent ways by which entities can be connected via structural and procedural links. The OPL, precisely defined by a grammar, is the textual counterpart modality of the graphical OPD set. OPL is a dual-purpose language, oriented towards humans as well as machines. Catering to human needs, OPL is designed as a constrained subset of English, which serves domain experts and system architects.

OPM consists of two types of elements: *entities* and *links*. Entities are classified into *things* and *states*. A thing is a generalisation of an object and a process. Objects are entities that exist, while processes are entities that transform things by generating, consuming, or affecting them. A state is a situation at which an object exists. Therefore, a state is not a stand-alone entity, but rather an entity that is *owned* by an object. At

any given point in time, the state-owning object is at one of its states. The status of an object, i.e., the current state of the object, is changed solely through an occurrence of a process. A *link* is an element that connects two entities to represent some semantic relation between them. Links can be *structural* or *procedural*. A structural link is a binary relation between two entities, which specifies a structural aspect of the modelled system, such as an aggregation-participation (whole-part) or a generalisation-specialization relation. A procedural link connects an entity with a process to denote a dynamic, behavioural flow of information, material, energy, or control. An *event link* is a specialization of a procedural link which models a significant happening in the system that takes place during a particular moment and might trigger a process if preconditions are met.

The basic assumption of OPM is that not only objects, but *objects and processes* constitute two equally-important classes of things, which together describe the functioning, structure and behaviour of a system in a single framework (i.e., without multiplying diagrams) in virtually any domain. OPM's basic principle is that structure and behaviour in a system are so intertwined that effectively separating them is extremely harmful, if not impossible. Therefore, unlike the object-oriented approach, behaviour in OPM is not necessarily encapsulated within a particular object class construct: using stand-alone processes, one can model a behaviour that involves several object classes and is integrated into the system structure. The behaviour of a system is manifested in three major ways: (i) processes can transform (generate, consume, or change) things (objects and processes), (ii) things can enable processes without being transformed by them, and (iii) things can trigger events that (at least potentially, if some conditions are met) invoke processes. Processes can be connected to the involved object classes through *procedural links*, which are divided, according to their functionality, into three groups: *enabling links*, *transformation links*, and *control links*.

The complexity of an OPM model is controlled through three *scaling processes*: *in-zooming/out-zooming*, in which the entity being refined is shown enclosing its constituent elements; *unfolding/folding*, in which the entity being refined is shown as the root of a directed graph; and state *expressing/suppressing*, which allows for showing or hiding the possible states of an object (see Chapter 12).

5.2.2 Meta-Modelling Languages for Processes

Research efforts are ongoing to define a unified meta-model of the process development addressed by methodologies, aimed at representing the existing methodologies in a uniform way, so as to promote their mutual comparison, their composition and reuse—this area is sometimes referred to as *Method Engineering* [14, 174].

SPEM (Software Process Engineering Meta-model, [200]) and OPF (Object-oriented Process, Environment and Notation Process Framework, [160]) are two key references for this purpose: as it could be expected, both were conceived for an object-oriented context, since most current methodologies adopt this paradigm as the reference one.

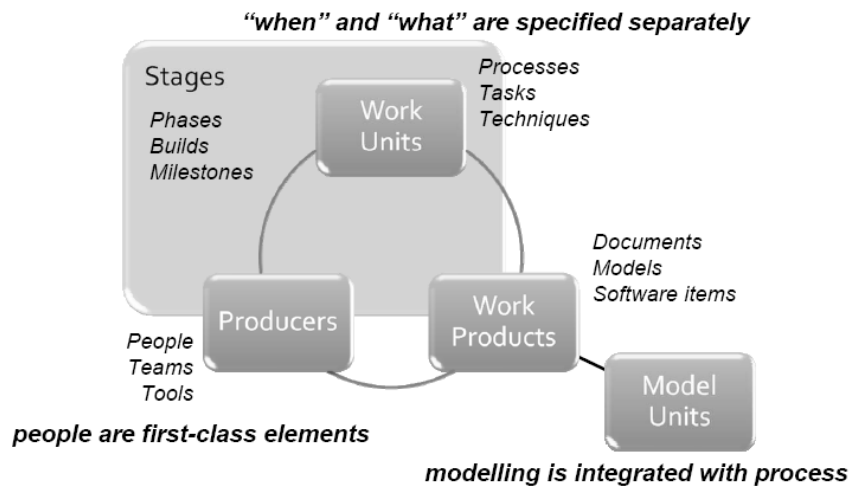


Figure 5.5: SPEM Overview

Subsection 5.2.2 and Subsection 5.2.2 present these two notations.

Software Process Engineering Meta-model

SPEM [200] version 1.1 is an OMG standard object-oriented meta-model defined as a UML profile and used to describe a concrete software development process or a family of related software development processes. SPEM is based on the idea that a software development process is a collaboration between active abstract entities called *roles* which perform operations called *activities* on concrete and real entities called *work products*. Each role interacts or collaborates by exchanging work products and triggering the execution of activities. The overall goal of a process is to bring a set of work products to a well-defined state (Figure 5.5).

Figure 5.6– from [200] – shows the main elements of the SPEM meta-model definition:

- *WorkProduct* is anything produced, consumed, or modified by a process. It may be a piece of information, a document, a model, source code, and so on
- *WorkProductKind* describes a category of work product, such as Text Document, UML Model, Executable, Code Library, and so on
- *WorkDefinition* is a kind of Operation that describes the work performed in the process. It can be decomposed reflexively:
 - *Activity* — describes a piece of work performed by one ProcessRole. An Activity may consist of atomic elements called Steps
 - *Phase* — is a specialization of WorkDefinition such that its precondition defines the phase entry criteria and its goal defines the phase exit criteria

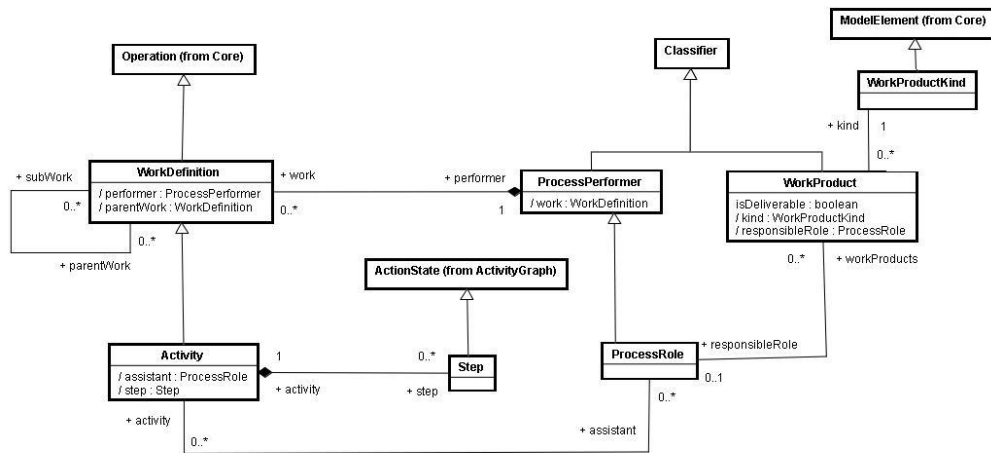


Figure 5.6: Process_Structure package of SPEM

- *Iteration* — An Iteration is a composite WorkDefinition with a minor phases
- *Lifecycle* — A process *Lifecycle* is defined as a sequence of Phases that achieve a specific goal. It defines the behaviour of a complete process to be enacted in a given project or program
- *ProcessPerformer* defines a performer for a set of WorkDefinitions in a process
- ProcessPerformer has a subclass, *ProcessRole*
- ProcessPerformer represents abstractly the whole process or one of its components, and is used to own WorkDefinitions that do not have a more specific owner
- *ProcessRole* defines responsibilities over specific WorkProducts, and defines the roles that perform and assist in specific activities
- *Guidance* provides more detailed information to practitioners about the associated ModelElement. I.e.: Technique is a kind of Guidance. A Technique is a detailed, precise algorithm used to create a work product

In addition SPEM provides a complete set of icons for the newly introduced concepts – the SPEM *notation* – that make it possible to build comprehensible models (Figure 5.7).

Object-oriented Process, Environment, and Notation

Object-oriented Process, Environment, and Notation (OPEN) [160] is a full lifecycle, process-focussed, methodological approach that was designed for the development of software intensive applications. OPEN is defined as a process framework, known as the OPF











Stereotype	Notation
WorkProduct	
WorkDefinition	
Guidance	
Activity	
ProcessRole	
ProcessPackage	
Phase	
Process	
Document	
UMLModel	

Figure 5.7: SPEM's icons

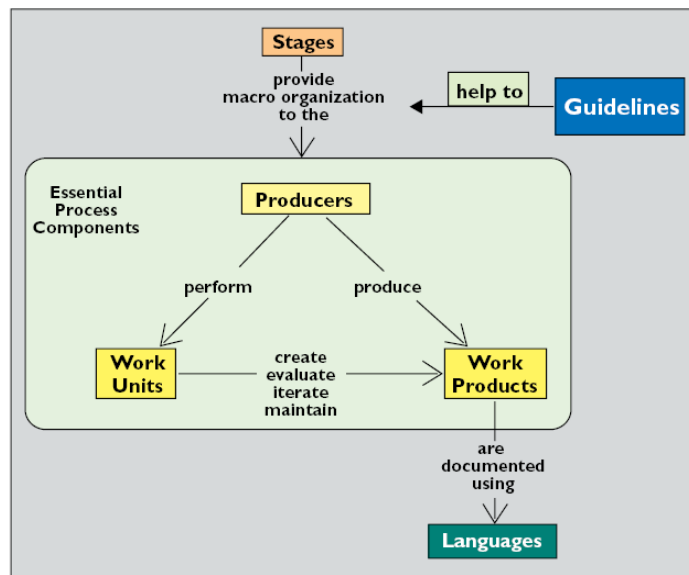


Figure 5.8: OPEN Meta-level classes

(OPEN Process Framework), that is a process meta-model from which can be generated an organisationally-specific process (instance). Each of these process instances is created by choosing specific Activities, Tasks and Techniques (three of the major metalevel classes) and specific configurations (Figure 5.8). The definition of process includes not only descriptions of phases, activities, tasks, and techniques but issues associated with human resources, technology, and the life-cycle model to be used. OPF leaves the choice of a notation to the developer, though suggesting UML as a good candidate.

The OPEN Process Framework provides a cohesive class library of predefined components that can be extended, instantiated and tailored for use on a specific project (Figure 5.8). These components are of the following five main types:

- *work product* is any significant thing of value (e.g., document, diagram, model, class, application) that is developed during a project
- *language* is the medium used to document a work product. Use case and object models are written using a modelling language such as the Unified Modelling Language (UML) or the OPEN Modelling Language (OML)
- *producer* is anything that produces (i.e., creates, evaluates, iterates, or maintains), either directly or indirectly, versions of one or more work products. The OPF distinguishes between those direct producers (persons as well as roles played by the people and tools that they use) and indirect producers (teams of people, organisations and endeavours)

- *work unit* is a functionally cohesive operation that is performed by a producer during an endeavour and that is reified as an object to provide flexibility during instantiation and tailoring of a process. The OPF provides the following predefined classes of work units:
 - *Task*— functionally cohesive operation that is performed by a direct producer. A task results in the creation, modification, or evaluation of a version of one or more work products
 - *Technique* — describes in full detail how a task is to be done
 - *Activity* — cohesive collection of workflows that produce a related set of work products. Activities in OPEN are coarse granular descriptions of what needs to be done
- *stage* is a formally identified and managed duration or a point in time, and it provides a macro organisation to the work units. The OPF contains the following predefined classes of stage:
 - *Cycle* — there are several types of cycle e.g. lifecycle
 - *Phase* — consisting of a sequence of one or more related builds, releases and deployments
 - *Workflow* — a sequence of contiguous task performances whereby producers collaborate to produce a work product
 - *Build* — a stage describing a chunk of time during which tasks are undertaken
 - *Release* — a stage which occurs less frequently than a build. In it, the contents of a build are released by the development organisation to another organisation
 - *Deployment* — occurs when the user not only receives the product but also, probably experimentally, puts it into service for on-site evaluation
 - *Milestone* — is a kind of Stage with no duration. It marks an event occurring

5.3 Meta-Modelling Languages for Infrastructures

To the best of our knowledge, in the infrastructures' research field there is not a standard language for modelling and representing an infrastructure. Taking inspiration from the traditional languages used in Software Engineering for representing methodologies' abstractions, UML [143] could be also used as a language for meta-modelling the infrastructures. In particular the UML's class diagrams are used for representing the static model (Subsection 5.3) and the UML's sequence diagrams for representing the dynamic model (Subsection 5.3).

UML was chosen as notation after a careful evaluation of other specification languages such as OPM [53] (see Subsection 5.2.1). However for the infrastructure the use of UML should not be misunderstood. UML is used only as a notation language for representing a high-level description of the infrastructures. This does not imply that the models represent the “real” infrastructure’s structure, and also does not imply an object-oriented implementation of the infrastructure.

In the reminder of this section the static and dynamic models which will be adopted in Chapter 10 for meta-modelling infrastructures are presented.

Static Models

The static model describes the structure of a system in terms of abstract elements and the relationships among them. The meta-model technique is chosen as a tool for representing the infrastructure’s static model. As for the methodologies, all infrastructures introduce some basic “run-time abstractions” (agents, organisations, resources, ...) in order to support deployment and execution of systems. So, it is not so strange to represent the relationships between such abstractions by means of a *meta-model*. The infrastructure’s meta-model becomes the key tool to compare an infrastructure with each other, and to check the consistency of an infrastructure when planning extensions or modifications. Meta-models are also an important guide for integrating different infrastructures avoiding several errors, such as assuming differences of concern when none exists, or assuming similarities of concern because of a common use of terms despite a different semantics. In addition, infrastructure meta-models could be used for associating each methodology’s concept to some suitable infrastructural abstraction(s): studying and comparing methodologies with infrastructures in terms of meta-models makes it possible to provide guidelines for mapping the design model of a methodology onto its implementation [124] (see Chapter 16).

Among the possible choices the UML class diagram is adopted as a tool for representing the meta-model. A class diagram is a type of static structure diagram that describes the structure of a system by showing the system’s classes, their attributes, and the relationships between the classes. Obviously the semantics associated to the UML’s elements is quite different from the “traditional” semantics used in the object-oriented context. For example in the infrastructure class diagram the “class elements” are not to be taken as the traditional object-oriented classes but simply as abstract entities supported by infrastructures.

Dynamic Models

The dynamic model describes and models the behaviour of the system at runtime. For the representation of this model different alternatives were evaluated such as OPM [53] and UML. OPM could be a good choice for representing the dynamic model, however OPM is

not well-known as UML, so as a tool for representing the infrastructure's dynamic model the UML sequence diagrams seems the best candidate.

As for the class diagram, the semantics associated to the language's elements is quite different from the "traditional" UML semantics. In particular, the only "interaction mechanism" supported by sequence diagram is message passing. In fact a sequence diagram provides a sequential map of messages passed between objects over time. So in the sequence diagrams that are shown in Chapter 10, the interactions among infrastructure entities are represented only by means of a message passing technique. This is obviously an abstraction over the real mechanisms adopted by each infrastructure. Each infrastructure, in fact, supports its own interaction mechanism: some infrastructures use message passing (like Jade), while others use different forms of mediated interaction (like **TuC-SoN**, **TOTA**,...). In a similar way, even if the interaction among abstract entities is illustrated with a synchronous semantics in the sequence diagrams, the interaction in the infrastructures does not necessarily occur in a synchronous way.

Another limitation of the sequence diagram is the difficulty to express "alternative paths" in the same diagram in order to have a global view of the interactions. For example, let us suppose we represent the interactions among three different entities A, B, C (Figure 5.9). The interaction initiator is the entity A that sends a message to B. B executes an internal elaboration and the result of this elaboration determines the receiver of the message that B will send. This is an example of "alternative paths": the interactions among entities depend on the result of internal elaboration. There are two possible ways to represent this kind of interaction, as illustrated in Figure 5.9. The first solution – top of the figure – represents each alternative path in a different diagram. Instead, the second solution – bottom of the figure – depicts the different paths in the same diagram with "explanatory notes" that explain the alternatives.

Even though both solutions can capture alternative path, each of them presents drawbacks. The first solution leads to an easier-to-read dynamic model than the second. However, the first solution also contains too many diagrams, so the general overview of the system behaviour is difficult to obtain. Instead, the second solution leads to a more compact model, which is however less readable, in particular when there are a lot of alternative paths.

The first solution seems good both for the better readability and for a possible future integration of all the different alternative diagrams relative to the same interaction in a unique Interaction Overview diagram. This diagram is supported in the UML version 2.0, but at the time of writing it is not so known as the sequence diagram and it is not so supported by the different UML CASE tools.

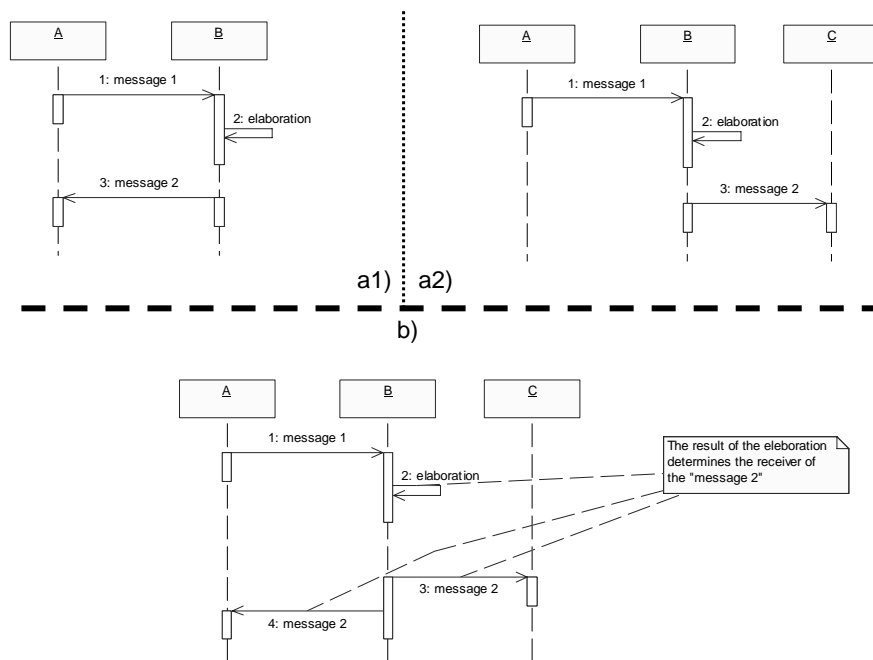


Figure 5.9: Problem in Sequence Diagram: two different diagram express the interaction (top), and the interaction in the same diagram (bottom)

5.4 Summing up

This chapter has presented the languages adopted for representing the different types of meta-models. For the meta-model of abstractions two different languages are proposed, UML and OPM. Both have strengths and drawbacks as illustrated in Section 5.2, however in the reminder of this thesis we have decided to use UML as a language for representing the meta-model for abstractions because UML is a standard and most known than OPM. Similar consideration are done for the meta-model languages for processes, where SPEM is our choice.

Finally, since there is not a standard for meta-modelling infrastructures, in the reminder of this thesis we adopted the method proposed in Section 5.3

6

AOSE & Meta-models

This chapter presents the abstractions and process meta-models for some of the most known AO methodologies presented in Chapter 4.

Since in the agent context there is not a universally accepted definition of agent nor it exists any very diffused model of the multi-agent system, the use of meta-models will help to clarify formally and compactly each methodology main building blocks and process. The importance of meta-modelling is not only for having a precise view of AO methodologies as a way to check their completeness and expressivity, or to compare them, but it is also useful to clarify the distance between AO methodologies and AO infrastructures current work. Expressing AO methodologies and AO infrastructures in formal meta-models presents an initial step to reduce the conceptual and technical gap amongst these two research areas. Consequently, there is the need to study the different AO methodologies, to compare their abstractions, rules, relationships, and the process they follow, and to have a comprehensive view of this variety of methodologies.

The remainder of this chapter is organised as follows: Section 6.1 presents the PASSI's meta-models, Section 6.2 presents the ADELFE's meta-models, Section 6.3 presents the Tropos's meta-models, and Section 6.4 presents the Gaia's meta-model. Finally Section 6.5 reports a summary of this chapter.

6.1 PASSI

This section provides a description of the PASSI meta-models. In particular Subsection 6.1.1 shows the concepts meta-model, while Subsection 6.1.2 depicts and high level description of the process meta-model.

6.1.1 PASSI: Concepts Meta-model

In the PASSI meta-model (Figure 6.1) [10], the Problem Domain deals with the user's problem in terms of *scenarios*, *requirements*, *ontology* and *resources*; scenarios describe a sequence of interactions among actors and the system.

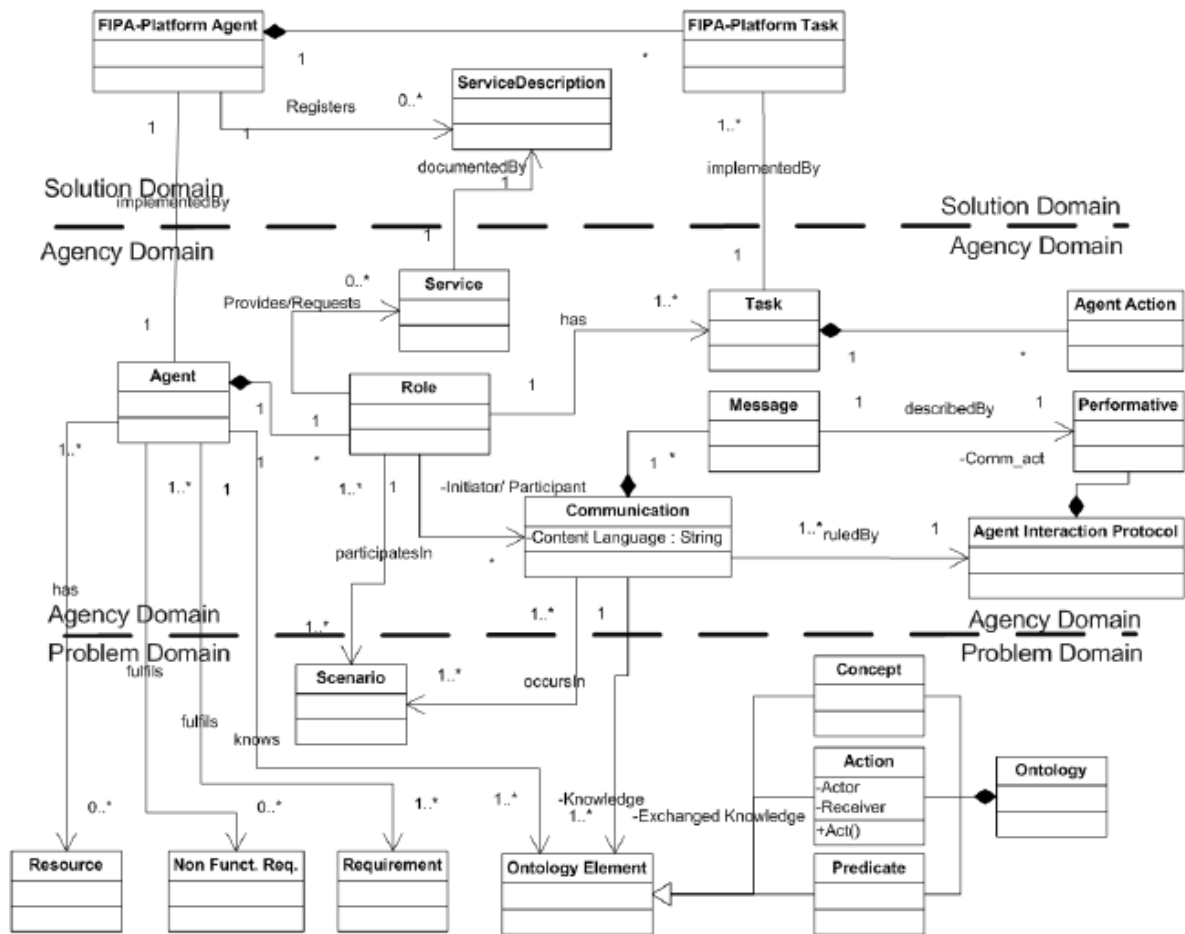


Figure 6.1: PASSI: concept meta-model

Requirements are represented with conventional use case diagrams. There is a strong point behind these choices: a lot of highly skilled designers are already present in different companies and can be more easily converted to the use of an agent-oriented approach if they are already confident with some of the key concepts used within it. Analysis related issues (like requirements and scenarios) being situated in the highest abstraction phase are strategic in enabling this skill reuse and allow a smooth entrance to the new paradigm. Ontological description of the domain is composed of *concepts* (categories of the domain), *actions* (performed in the domain and effecting the status of concepts) and *predicates* (asserting something about a portion of the domain). This represents the domain in a way that is substantially richer than the classic structural representations produced in the object-oriented analysis phase. As an example, we can consider ontologies devoted to reasoning about strategies or problem solving methods whose essence is very difficult to capture in object-oriented structures. *Resources* are the last element of the problem domain. They can be accessed/ shared/manipulated by agents. A resource could be

a repository of data (like a relational database), an image/video or also a good to be sold/bought. We prefer to explicitly model them since goals of most systems are related to using and capitalizing on available resources.

The Agency Domain contains the elements of the agent-based solution. None of these elements is directly implemented; they are converted to the correspondent object-oriented entity that constitutes the real code-level implementation. The concept of *agent* is the real center of this part of the model; each agent in PASSI is responsible for realising some functionalities descending from one or more requirements. The direct link between a requirement and the responsible agent is one of the strategic decisions taken when conceiving PASSI. Each agent during its life plays some *roles* which are portions of the agent social behaviour characterised by some specificity such as a *goal*, or providing a *functionality/service*. From this definition it is clear that roles could use communications in order to realise their relationships or portions of behaviour (called *tasks*) to actuate the role proclivity. In PASSI, the term task is used to mean an atomic part of the overall agent behaviour and, therefore, an agent can accomplishing its duties by composing the set of its own tasks. Tasks cannot be shared among agents, but their capabilities could be offered by the agent to the society as services (often a service is obtained composing more than one task); obviously according to agent autonomy, each single agent has the possibility of accepting or refusing to provide a service if this does not match its personal attitudes and will. A communication is composed of one or more *messages* expressed in an encoding language (e.g. ACL [62]) that is totally transparent to agents.

Finally, the Implementation Domain describes the structure of the code solution in the chosen FIPA-compliant implementation platforms and it is essentially composed of three elements: (i) the *FIPA-Platform Agent* that represents the implementation class for the agent entity represented in the Agency domain; (ii) the *FIPA-Platform Task* that is the implementation structure available for the agent's task and, finally, (iii) the *Service* element that describes a set of functionalities offered by the agent under a specific name that is registered in the platform service directory and therefore can be required by other agents to reach their goals.

6.1.2 PASSI: Process Meta-model

The PASSI process modelled by SPEM is composed of five disciplines: system requirements, agent society, agent implementation, code and deployment [121]. In Figure 6.2 the PASSI process is shown in terms of the composing phases and their workproducts. Each phase produces a document that is usually composed aggregating the UML models and work products produced during the related activities. Each phase is composed of one or more sub-phases, each one responsible for designing or refining one or more artefacts that are parts of the corresponding model (for instance the System Requirements model includes an agent identification diagram that is a kind of UML use case diagrams but also some text documents like a glossary and the system use scenarios).

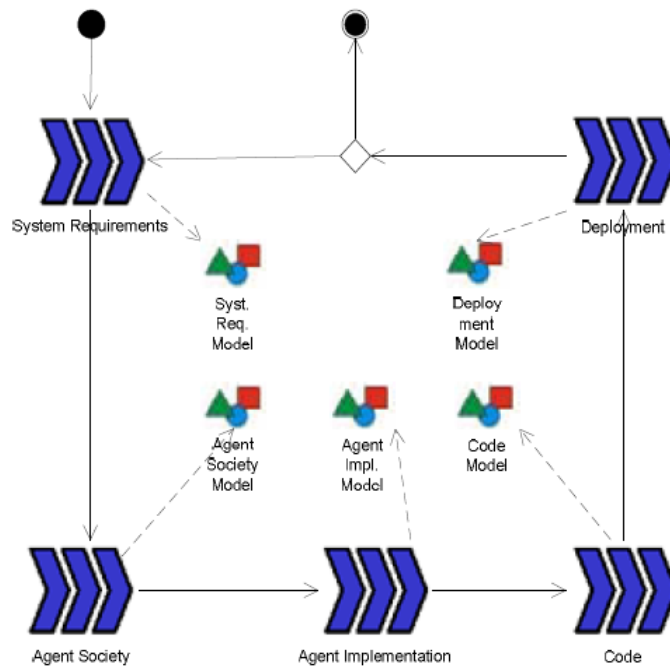


Figure 6.2: PASSI Process

The system requirements discipline covers all the phases related to requirement elicitation, analysis and agents/roles identification (Figure 6.3).

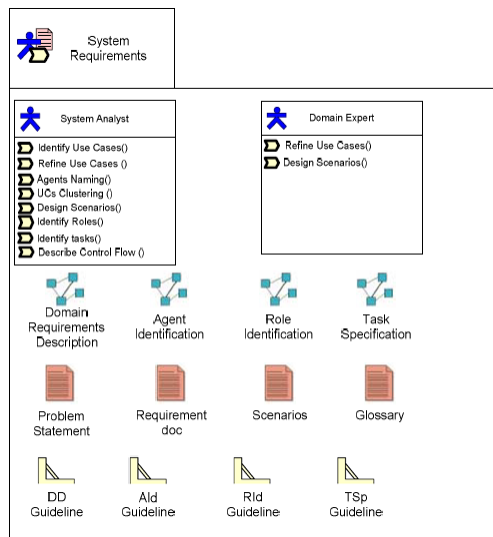


Figure 6.3: PASSI: system requirement

The agent society discipline faces all the aspects of the agent society: ontology, communications, roles description, interaction protocols (Figure 6.4)

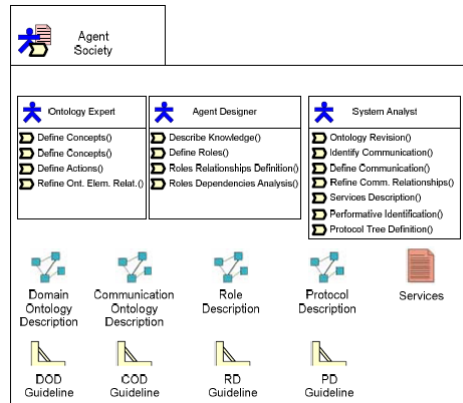


Figure 6.4: PASSI: agent society

The agent implementation discipline provides a view on the system architecture in terms of classes and methods to describe the structure and the behaviour of single agent (Figure 6.5).

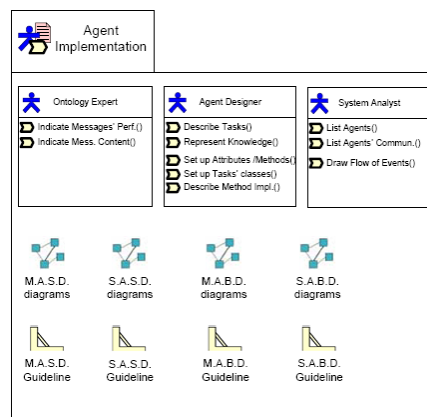


Figure 6.5: PASSI: agent implementation

The code discipline provides a library of class and activity diagrams with associated reusable code and source code for the target system (Figure 6.6 part a)).

Finally the deployment discipline depicts how the agents are deployed and which constraints are defined/identified for their migration and mobility (Figure 6.6 part b)).

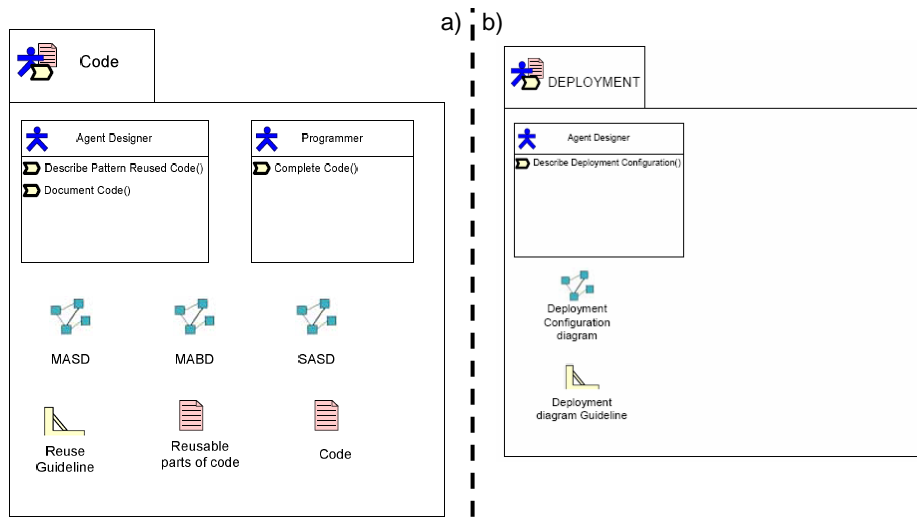


Figure 6.6: PASSI: code (a) and deployment (b)

6.2 ADELFE

This section provides a description of the ADELFE meta-models. In particular Subsection 6.2.1 shows the concepts meta-model, while Subsection 6.2.2 depicts a high level description of the process meta-model.

6.2.1 ADELFE: Concepts Meta-model

The meta-model adopted for ADELFE (Figure 6.7) is fundamentally explained by AMAS theory and by the features a cooperative agent possesses [10].

Local cooperation rules enable it to detect and solve *Non Cooperative Situations* (NCS). These NCS are cooperation failures that are, from its point of view, inconsistent with its cooperative social attitude. Different kinds of such failures can be detected according to the context of the concerned application, such as *Incomprehension* (an agent does not understand a perceived signal), *Ambiguity* (it has several contradictory interpretations for a perceived signal), *Incompetence* (it cannot satisfy the request of another), *Unproductiveness* (it receives an already known piece of information or some information that leads to no reasoning for it), *Concurrency* (several agents want to access an exclusive resource), *Conflict* (several agents want to realise the same activity) or *Uselessness* (an agent may make an action that is not beneficial, according to its beliefs, to other agents). When detecting a NCS, an agent does all it is able to do to solve it to stay cooperative for others. For example, faced with an incomprehension situation, it does not ignore the message but will transmit it to agents that seem (from its point of view) relevant to deal with it. An agent possesses world representations that are beliefs concerning other agents,

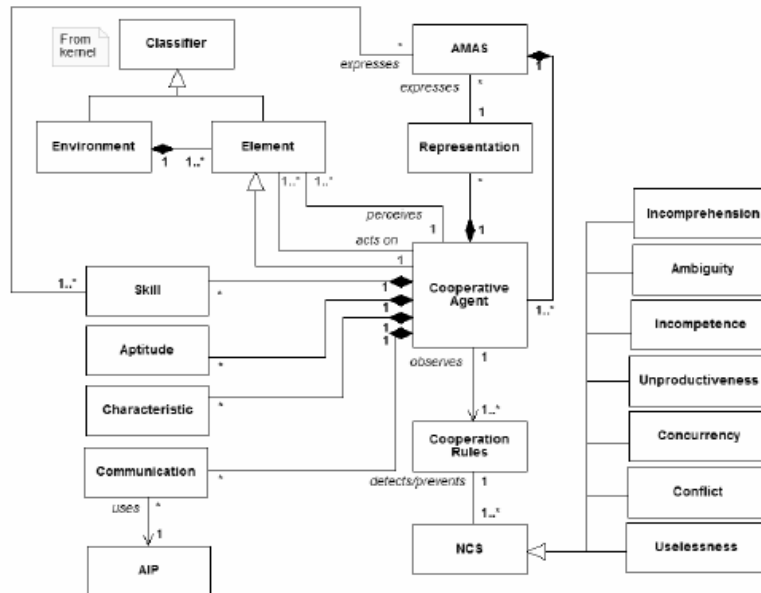


Figure 6.7: ADELFE: concept meta-model

the *physical environment* or the agent itself. These representations are used by the agent to determine its behaviour. If an agent has representations that may evolve (e.g., a semantic network), these representations can be expressed using a multi-agent system. A representation can be shared by different agents.

An agent is able to communicate with other agents or its environment. This communication can be done in a direct manner (by exchanging messages) or an indirect one (through the environment). Tools that enable an agent to communicate are interaction languages. When an agent uses a direct communication through message exchanges, agent interaction protocols may also be used to express the communication pattern between agents. An agent can interact with its environment (physical or social) by means of perceptions and actions. For an agent, an action is a way to act on its environment during its action phase and a perception enables it to receive information from this environment. *Aptitudes* show the ability of an agent to reason both about knowledge and beliefs it owns. For instance, an aptitude of a software agent can be expressed by an inference engine on a base of rules or any other processing on perceptions and world representations. Aptitudes can also be expressed using data, e.g. an integer value which represents the exploration depth of a planning tree. An agent owns some *skills* that are specific knowledge that enable it to realise its own partial function. For instance, a skill may be a simple datum which is useful to act on the world (e.g., an integer distance which represents the minimal distance a robot has to respect to avoid obstacles) or may be more complex when

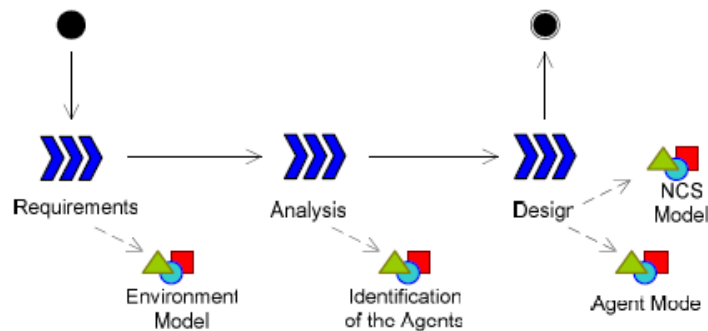


Figure 6.8: ADELFE Process

expressing reasoning that the agent makes during its decision phase (e.g., a reasoning to avoid obstacles). If they are complex and able to evolve, skills may also be implemented by MAS.

An agent may possess some *characteristics* which are its intrinsic or physical properties. It may be, for instance, the size of an agent or the number of legs of a robot-like or ant-like agent. A characteristic may also be something the agent can perform to modify or update one of its properties; for example, if the agent is an ant, enabling it to modify its number of legs.

6.2.2 ADELFE: Process Meta-model

The ADELFE process modelled by SPEM is composed of three disciplines: requirements, analysis and design [1]. In Figure 6.8 the ADELFE process is shown in terms of the composing phases and their workproducts:

- Requirements phase produces Functional Description model, Scenarios, Interface Models, Environment Definition, Requirement Set, Keyword Set and Usage Interface Prototypes.
- Analysis phase produces Domain Model, AMAS Adequacy Synthesis, Software Architecture, Environment Definition, Internal Interaction between Domain Classes.
- Design phase produces the Detailed Architecture, Interaction Languages, Protocols Diagram and Design Model.

During the requirements phase (Figure 6.9) it is necessary to give an Environment Model that, in the AMAS theory, will serve as base for the process of adaptation. This process begins with the interactions between the system and the environment. The environment model includes the following activities: actors determination, context definition and environment characterisation.

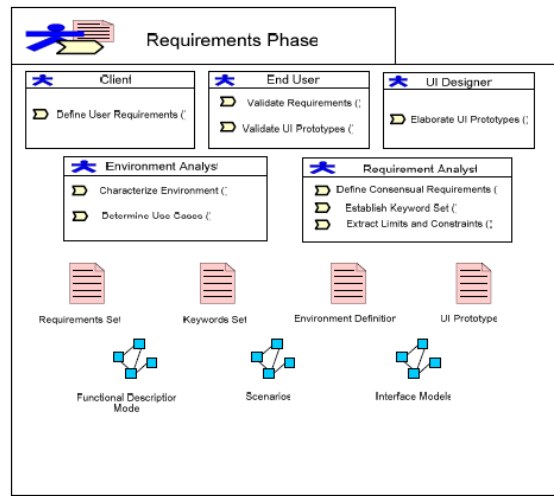


Figure 6.9: ADELFE Requirements Phase

In the analysis phase (Figure 6.10), the previously defined entities are analyzed in order to specify which will be agents. An agent is an entity having the capability to evolve during an unexpected situation, showing new behaviours and skills. Two activities are added to the classical RUP: the agent identification and the adequacy at the AMAS theory.

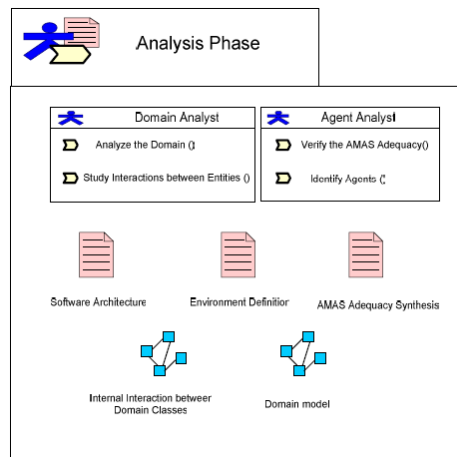


Figure 6.10: ADELFE Analysis Phase

The design phase (Figure 6.11) aims to define the agent’s architecture describing their behaviours; the result of this activity adds two models to the RUP: the Agent Model and the Non Cooperative Situations (NCS) Model.

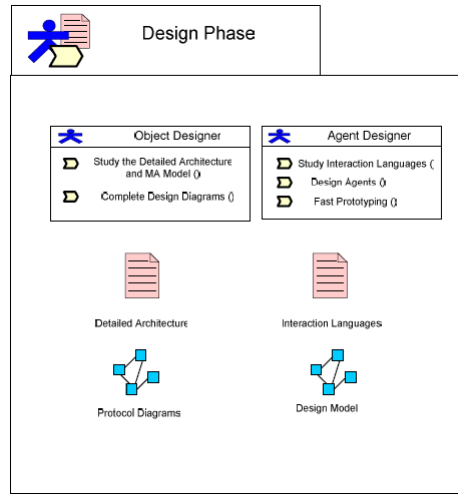


Figure 6.11: ADELFE Design Phase

6.3 Tropos

This section provides a descriptions of the Tropos meta-models. In particular Subsection 6.3.1 shows the concepts meta-model, while Subsection 6.3.2 depicts a high level description of the process meta-model.

6.3.1 Tropos: Concepts Meta-model

Tropos adopts extended i^* notation with *actor*, *goal*, *task/plan*, *softgoal*, *resource*, and *dependency* as basic modelling constructs. These concepts allow one to model both software systems and organisations, and are used through the whole software development process, from early requirements down to implementation. A goal represents the strategic interests of an actor. A softgoal, as opposed to a *hardgoal* (or simply a goal), is a goal that is typically a nonfunctional attribute or quality, with no clear-cut criteria as to when it is achieved. A task/plan specifies a particular course of action that produces a desired effect, and can be executed in order to satisfy a goal. Goals and tasks can be related to softgoals through qualitative relationships (labelled “+” and “-”) to indicate that the goal/task contributes positively or negatively to the fulfillment of the softgoal. A resource represents a physical or an informational entity. Finally, a dependency between two actors indicates that one actor depends on another to accomplish a goal, execute a task/plan, or deliver a resource. Tropos modelling approach includes two types of diagrams, namely, actor and goal diagrams. An actor diagram is a graph with actors (agents, positions, or roles) as nodes, and dependencies among actors as edges. A goal diagram is used for

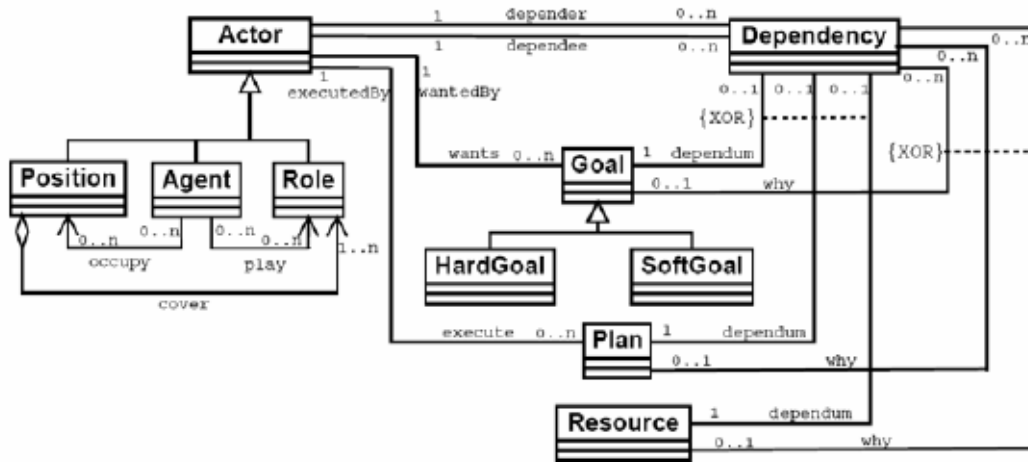


Figure 6.12: Tropos: basic concept meta-model

supporting the means-ends analysis conducted by each actor as it attempts to ensure that its goals will eventually be fulfilled (through goal decompositions and delegations).

In Figure 6.12 the portion of the Tropos meta-model taken from [31] with the basic modelling constructs is shown. Agent, role and position are specializations of the concept of actor. A position can cover 1..n roles, whereas an agent can play 0..n roles and occupy 0..n positions. An actor can have 0..n goals, which can be both hard and softgoals, and each of them wanted by 1 actor. An actor dependency relates a depender, dependee, and dependum (i.e. goal, task, or resource). It is possible to specify also a reason for the dependency (labelled as why). The concepts related to the Tropos goal diagram are depicted in Figure 6.13. As follows from the diagram, goals can be analyzed, from the point of view of an actor, by means-end analysis, contribution analysis and boolean decomposition. Means-end analysis allows one to specify the means (goals/plans/resources) that are used in order to achieve the end (a goal). Contribution analysis aims at identifying goals that can contribute positively or negatively towards the fulfillment of other goals. Decomposition defines a generic boolean decomposition of a root goal into AND- or OR-subgoals. Means-end analysis and AND/OR decomposition can be applied to plans also.

6.3.2 Tropos: Process Meta-model

The Tropos process modelled by SPEM is composed of five disciplines: early requirements, late requirements, architectural design, detailed design and implementation [121]. In Figure 6.14 the Tropos process is shown in terms of the composing phases and their workproducts.

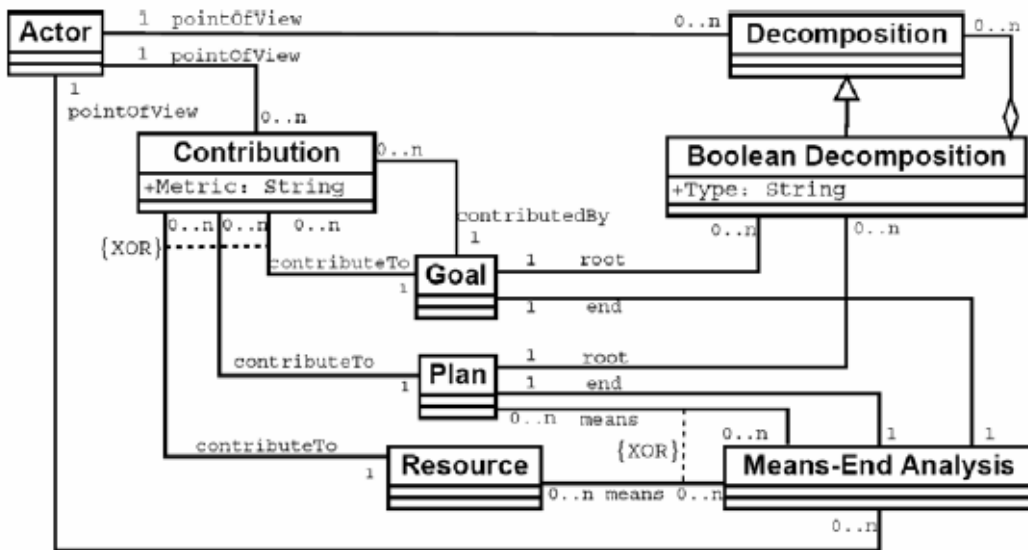


Figure 6.13: Tropos: concepts of goal diagram

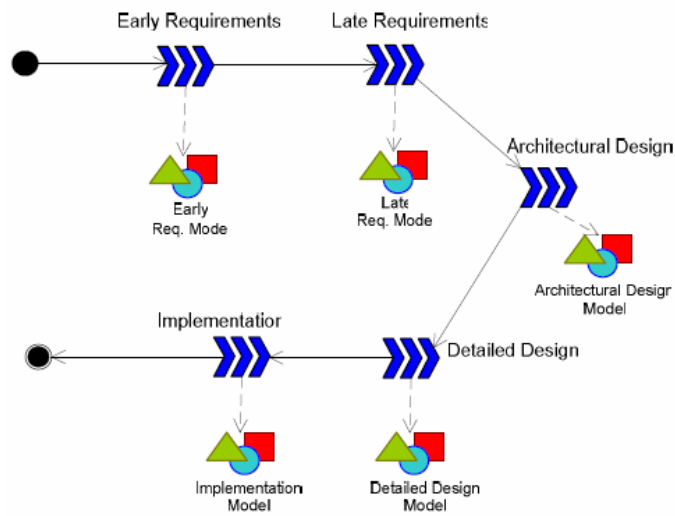


Figure 6.14: Tropos Process

The early requirements discipline shows stakeholders and their goals, dependencies and resources (Figure 6.15).

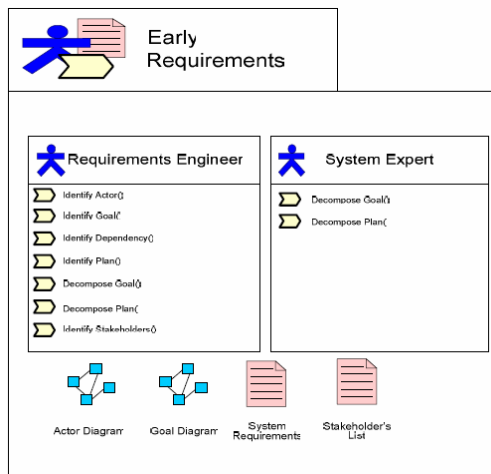


Figure 6.15: Tropos: early requirement

The late requirements discipline shows the “system-to-be” as new actor and its goals, dependencies and resources (Figure 6.16).

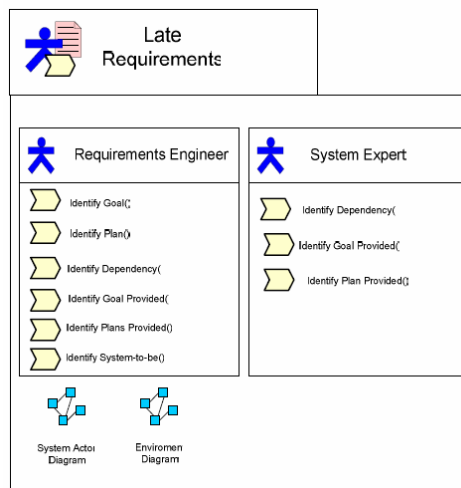


Figure 6.16: Tropos: late requirement

The architectural design discipline defines the system architecture in terms of sub-actors and their relatives goals, dependencies and relations. In addition the software agents related to sub-actors are defined (Figure 6.17).

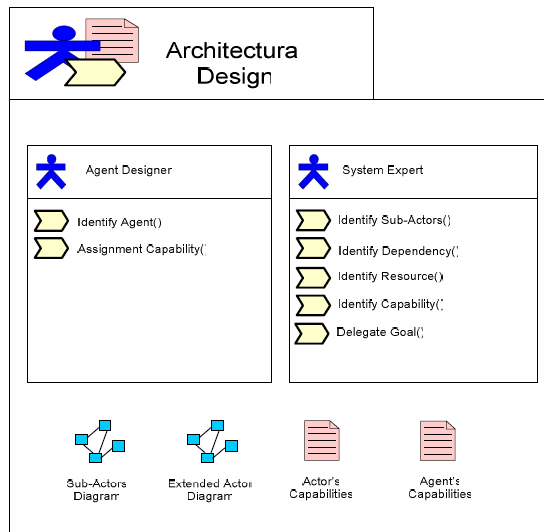


Figure 6.17: Tropos: architectural design

The detailed design discipline covers the agent design in terms of capabilities, beliefs, goals and plans (Figure 6.18).

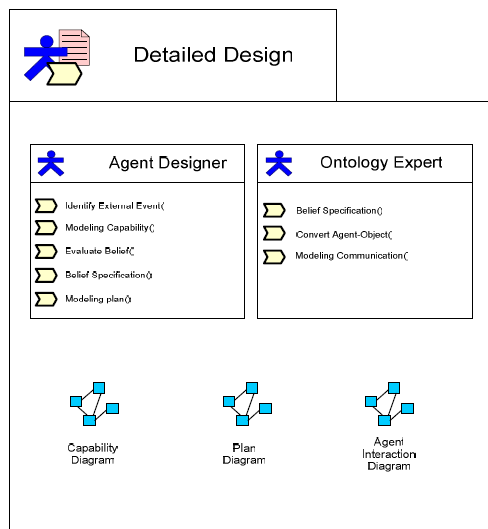


Figure 6.18: Tropos: detailed design

The implementation discipline creates a code skeleton of the detailed design specifications (Figure 6.19).

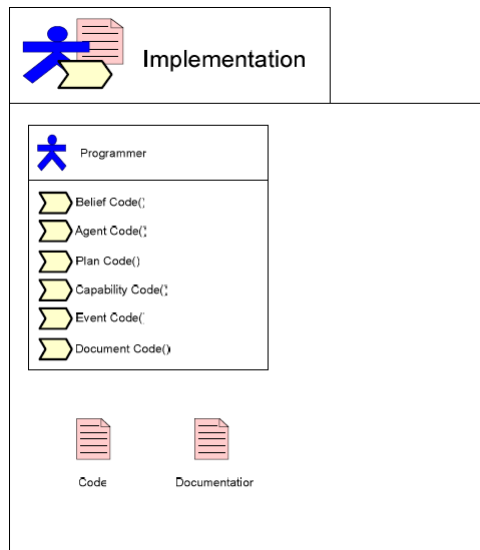


Figure 6.19: Tropos: implementation

6.4 Gaia

This section provides a description of the Gaia concept meta-model. The process meta-model is not reported because it is currently not available for the new version of 2003 [229]. The process meta-model of the previous version [225] could be found in [121].

The meta-model of GAIA taken is depicted in Figure 6.20.

Organisations are viewed in GAIA as collections of *roles*, which are defined in terms of *responsibilities*, *permissions*, *activities* and *protocols*. Responsibilities define the functionality of the role, while permissions are the “rights” which allow the role to perform its responsibilities. Activities are computations that can be executed by the role along, and protocols define the interaction between roles. As soon as the complexity of systems increases, modularity and encapsulation principles suggest dividing the system into different sub-organisations, with a subset of the agents being possibly involved in multiple organisations.

In each organisation, an agent can play one or more roles, which defines what it is expected to do in the organisation, both in concert with other agents and in respect to the organisation itself. To accomplish their roles, agents typically need to interact with each other to exchange knowledge and coordinate their activities. These interactions occur according to patterns and *protocols* dictated by the nature of the role itself. In addition, a MAS is typically immersed in an environment with which the agents may need to interact in order to accomplish their roles. That portion of the environment that agents can sense and effect is determined by the agent’s specific role, as well as by its current status. Identifying and modelling the environment involves determining all the

entities and *resources* that the MAS can exploit, control, or consume when it is working towards the achievement of the organisational goal.

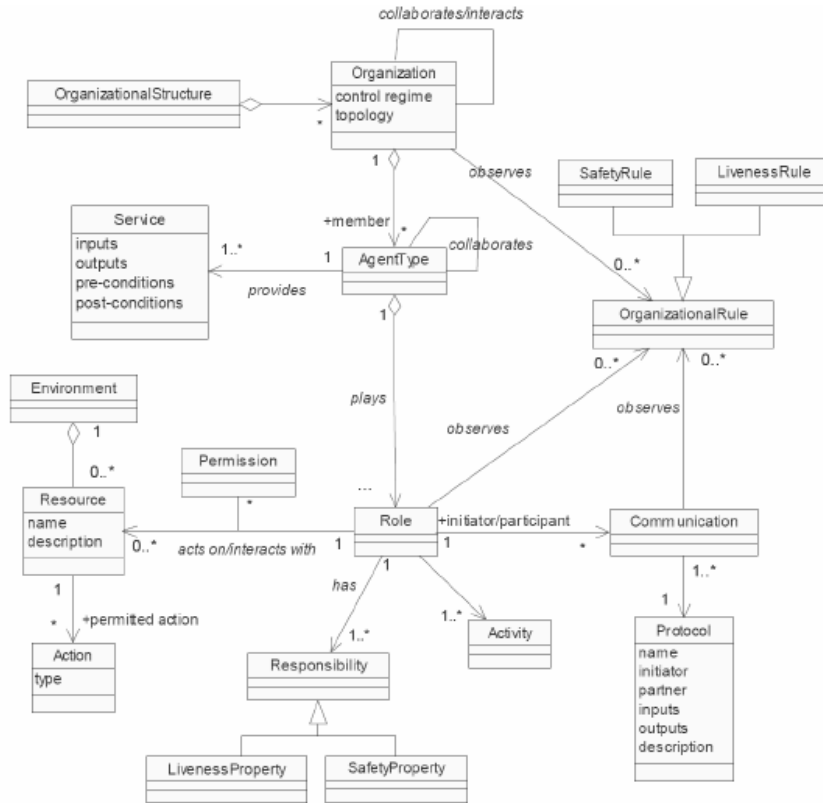


Figure 6.20: Gaia: concept meta-model

However, although role and interaction models can be useful to fully describe an existing organisation, they are of limited value in building an organisation. The organisational structure is not implicitly defined via the role model, instead the identification of the roles is explicitly derived from an analysis of the chosen organisational structure. As a consequence the role model and the related interaction model will be completely defined in the design phase when an accurate identification of the organisational structure will take place. This motivates the introduction of the notions of *organisational rules* and *organisational structures*. It is possible to distinguish between safety and liveness organisational rules. The former refer to the invariants that must be respected by the organisation for it to work coherently; the latter express the dynamics of the organisation. A role model implicitly defines the topology of the interaction patterns and the control regime of the organisations activities. That is, it implicitly defines the overall architecture of the MAS organisation, i.e. its organisational structure.

6.5 Summing up

This chapter has presented the meta-models of some most known AO methodologies. As mentioned above a meta-model addresses all of the different aspects of methodologies – i.e., the process to follow and the abstractions adopted by methodologies – and it is necessary for studying the completeness and the expressiveness of a methodology, and when comparing different methodologies. These characteristics are important and make meta-modelling techniques very appealing in the the context of AOSE where there is a incredible proliferation of methodologies.

However, the power of the meta-modelling is not only limited to the study and the comparison of the methodologies. In fact, the use of meta-modelling techniques allows designers to combine fragments [36] – i.e., pieces of methodologies – of existing AO methodologies for obtaining a new methodology that satisfies the requirements of a specific application domain (Subsection 4.3.3). This method is called Method Engineering [15, 16] that is the engineering discipline to design, construct and adapt methodologies, techniques and tools for the development of information systems. Similarly as software engineering is concerned with all aspects of software production, method engineering deals with all engineering activities related to methodologies, techniques and tools.

The assembly of a new methodology starting from the fragments of the existing methodologies – or fragments built ad hoc – is not a new idea. The study started in the early nineties in the object-oriented field [16, 84, 185] and then it was introduced in the agent-oriented field in 2003 by the IEEE-FIPA Methodology Technical Committee (FIPA Foundation for Intelligent Physical Agents) [121]. This group have both defined the method fragments for AO methodologies and developed a technique for the fragments integration [36] and its relative tool [18].

So the study of the existing methodologies’s meta-models – for abstractions and for processes – becomes the key tool for enabling the construction of new methodologies.

7

The Agents & Artifacts Meta-Model

This chapter presents a new conceptual framework – called Agents&Artifacts – for modelling and designing agent-oriented systems. According to social / psychological theories like Activity Theory (AT) [137], artifacts plays a fundamental role in the context of human organisations for supporting cooperative work and, more generally, complex collaboration activities. Artifacts are either physical or cognitive tools that are shared and exploited by the collectivity of individuals for achieving individual as well as global objectives. The conceptual framework of artifacts for MAS is meant to bring the same sort of approach to MAS. The adoption of artifacts in the MAS context changes the traditional view of the MAS environment: from a simple deployment context MAS environment is transformed to a new MAS design dimension.

Then agents and artifacts become the new “ingredients” for engineering agent-oriented systems. More precisely, agents are the basic abstractions to represent active, task-/goal-oriented components, designed to pro-actively carry out one or more activities towards the achievement of an objective, requiring different levels of skill and reasoning capabilities. On the other hand, artifacts are the basic abstractions to represent passive, function-oriented building blocks, which are constructed and used by agents, either individually or cooperatively, during their working activities. So agents can be used to model individual activities, while artifacts can be well suited for mediating the interaction between individual components and their environment (including the other components), and for embodying the portion of the environment that is explicitly designed to support agents’ activities [146].

The remainder of this chapter is organised as follows: Section 7.1 presents the general view of the A&A meta-model, Section 7.2 presents artifacts and their characteristics / features, and Section 7.3 presents the concept of workspace, the third ingredient of the A&A meta-model. Finally Section 7.4 summarises the chapter.

7.1 A&A Meta-Model

Agents&Artifacts (A&A) is a novel conceptual framework introduced in the context of AOSE for modelling and designing agent-based software systems [146, 152, 153]. A&A

introduces three basic kinds of general-purpose abstractions to understand and model complex systems [130]:

- *agents*: pro-active components of the systems, encapsulating the autonomous execution of some kind of activity inside some sort of environment.
- *artifacts*: passive components of the systems such as resources and media that are intentionally constructed, shared, manipulated and used by agents to support their activities, either cooperatively or competitively (Section 7.2).
- *workspaces*: logical containers of agents and artifacts, useful for defining the topology for the environment and providing a way to define a notion of locality (Section 7.3).

So one fundamental difference with respect to existing agent-based models is the adoption of artifacts as a *first-class abstraction* also to model and design those parts of the MAS which are function-oriented, i.e. designed to provide some kind of functions [130].

One of the key issues of in the A&A approach is how artifacts can be effectively exploited to improve agent ability to achieve individual as well as social goals [152]. The main questions to be answered are then: How should agents reason to use artifacts in the best way, making their life simpler and their action more effective? How can agents reason to select artifacts to use? How can agents reason to construct or adapt artifact behaviour in order to fit their goals?

On the one hand, the simplest case concerns agents directly programmed to use specific artifacts, with usage protocols directly defined by the programmer either as part of the procedural knowledge / plans of the agent for goal-governed systems, or as part of agent behaviour in goal-oriented system. In spite of its simplicity, this case can bring several advantages for MAS engineers, exploiting separation of concerns for programming simpler agents, by imposing some burden upon specifically-designed artifacts. On the other hand, the intuition is that in the case of fully-open systems, the capability of the artifact to describe itself, its function, interface, structure and behaviour could be the key for building open MASs where intelligent agents dynamically look for and select artifacts to use, and then exploit them for their own goals.

At first glance, it seems possible to frame the agent ability to use artifacts in a hierarchy, according to five different cognitive levels at which the agent can use an artifact [152, 153]:

unaware use at this level, both agents and agent designers exploit artifacts without being aware of it: the artifact is used implicitly, since it is not denoted explicitly. In other words, the representation of agent actions never refers explicitly to the execution of operation on some kind of artifact.

embedded / programmed use at this level, agents use some artifacts according to what has been explicitly programmed by the designer: so, the artifact selection is explicitly made by the designer, and the knowledge about its use is implicitly encoded by the designer in the agent. In the case of cognitive agents, for instance, agent designers can specify usage protocols directly as part of the agent plan. From the agent point of view, there is no need to understand explicitly artifact operating instructions or function: the only requirement is that the agent model adopted could be expressive enough to model in some way the execution of external actions and the perception of external events.

cognitive use at this level, the agent designer directly embeds in the agent knowledge about what artifacts to use, but how to exploit the artifacts is dynamically discovered by the agent, reading the operating instructions. Artifact selection is still a designer affair, while how to use it is delegated to the agent's rational capabilities. So, generally speaking the agent must be able to discover the artifact function, and the way to use it and to make it fit the agent goals. An obvious way to enable agent discovery is to make the artifact explicitly represent their function, interface, structure and behaviour.

cognitive selection and use at this level, agents autonomously select artifacts to use, understand how to make them work, and then use them: as a result, both artifact selection and use are in the hands of the agents. It is worth noting that such a selection process could also concern sets of cooperative agents, for instance interested in using a coordination artifact for their social activities.

construction and manipulation at this level, agents are lifted to the role of designers of artifacts. Here, agents are supposed to understand how artifacts work, and how to adapt their behaviour (or to build new ones from scratch) in order to devise a better course of actions toward the agent goals. For its complexity, this level more often concerns humans: however, not-so-complex agents can be adopted to change artifact behaviour according to some schema explicitly pre-defined by the agent designers.

To enable such scenarios, proper models, theories and then supporting frameworks are needed, making artifacts first-class entities from design to runtime.

7.2 Artifacts

The sources for a theory of artifacts can be found in a number of different research fields, ranging from organisational / psychological theories – such as Activity Theory (AT) [137] – to anthropology [75, 89], and obviously including the area of coordination models [159]. In particular, AT is based on a structured model that constitutes an activity, and on the

mediating role of artifacts. Any activity is characterised by a subject, an object and by one or more mediating artifacts:

- a subject is an agent or group engaged in an activity;
- an object (in the sense of objective) is held by the subject and motivates the activity, giving it a specific direction (the objective of the activity); the object of activity can be a wide variety of things, from mental objectives (e.g., making a plan) as well as physical ones (e.g., writing a paper);
- the mediation artifacts, which are the tools that enable and mediate subject actions toward the object of the activity. The mediating artifacts can be either physical or abstract / cognitive; examples are: symbols, rules, operating procedures, heuristics, scripts, individual / collective experiences, and languages.

According to AT, mediating tools have both an *enabling* and a *constraining* function: on the one hand, they 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. Mediating artifacts shape the way human beings interact with reality [176]. Given this definition, artifacts could be fruitfully adopted as enabling and constraining tools also in the agent paradigm, where MASs are built around the concepts of society of agents. So, two basic aims can be immediately identified in the artifact abstraction:

- social (constructive): an artifact is an abstraction essential for constructing social activities, creating the agent interaction space;
- normative (regulatory): an artifact is an abstraction essential for ruling social activities, ruling the agent interaction space.

So, artifacts functions as media enabling agent interaction (such as coordination artifacts [153]) play a key role in MAS, being what actually shapes the interaction space among the agent. So, changing the behaviour of an artifact working as medium could have a strong impact on the overall system behaviour.

In particular artifacts allow agents to use the same cognitive level when they interact with other agents and with environment. Usually, agents speak with each other via languages like FIPA-ACL (agent communication language) [62], while exploiting the environmental resources as a means for lower-level interaction. So, the agents' *interaction space* spans over different cognitive levels, because agents cannot interact with the other components (agents and resources) of the MAS in a uniform way [188]. Artifacts, instead, make it possible to wrap the resources of a MAS and bring them to the cognitive level of agents, so that both the interaction among agents (on the one side) and between agents and artifacts (on the other) can occur at the same cognitive level, exploiting the high-level

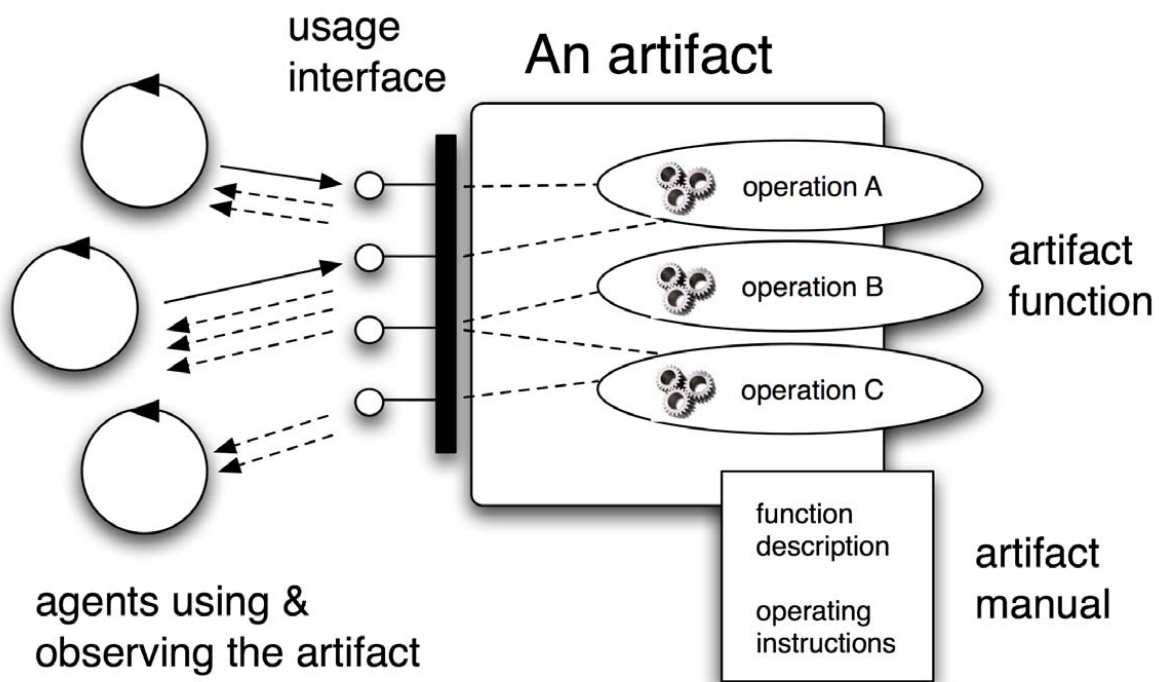


Figure 7.1: An abstract model of the artifact [180]

language that describes artifacts' services as the common language. In order to make this possible, the language should be standard enough to support both the MAS openness and agent mobility, allowing heterogeneous agents to join a MAS, discover and use the services provided by the artifacts. By the way, this choice also simplifies the design of the MAS interaction space, since engineers no longer need to design ad-hoc protocols for each resource in the environment (the relationship between each resource and the artifact that wraps it concerns the internal design of the artifact and does not affect the interaction space of the MAS).

A more detailed characterisation of the artifact abstraction, in terms of fundamental properties and features, can be found in Subsection 7.2.1, while Subsection 7.2.2 outlines a possible taxonomy for artifacts.

7.2.1 Features

According to [180], each artifact type is to be equipped by the artifact designer with a *manual* composed essentially by the (i) a usage interface, (ii) operating instructions, and (iii) a function description (Figure 7.1).

Usage Interface — One of the core differences between artifacts and agents, as computational entities populating a MAS, lies in the concept of *operation*, which is the

means by which an artifact provides for a function [152]. An agent executes an action over an artifact by invoking an artifact operation. Execution possibly terminates with an operation completion, typically representing the outcome of the invocation, which the agent comes to be aware of in terms of perception. The set of operations provided by an artifact defines what is called its *usage interface*, which (intentionally) resembles interfaces of services, components or objects in the object-oriented sense of the term. In MASs, this interaction schema is peculiar to artifacts, and makes them intrinsically different from agents. While an agent has no interface, acts and senses the environment, encapsulates its control, and brings about its goals proactively and autonomously, an artifact has instead a usage interface, is used by agents (and never the opposite), is driven by their control, and automates a specific service in a predictable way without the blessing of autonomy. Hence, owning an interface strongly clearly differentiates agents and artifacts, and is therefore to be used by the MAS engineer as a basic discriminative property between them (Figure 7.1).

Operating Instructions — Coupled with a usage interface, an artifact could provide agents with *operating instructions* [152]. Operating instructions are a description of the procedure an agent has to follow to meaningfully interact with an artifact over time. Most remarkably, one such description is history dependent, so that actions and perceptions occurring at a given time may influence the remainder of the interaction with the artifact. Therefore, operating instructions are basically seen as an exploitation protocol of actions / perceptions. This protocol is possibly furthermore annotated with information on the intended preconditions and effects on the agent mental state, which a rational agent should read and exploit to give a meaning to operating instructions. Artifacts being conceptually similar to devices used by humans, operating instructions play a role similar to a manual, which a human reads to know how to use the device on a step-by-step basis, and depending on the expected outcomes he/she needs to achieve (Figure 7.1).

Function Description — Finally, an artifact could be characterised by a *function description* [152]. This is a description of the functionality provided by the artifact, which agents can use essentially for artifact selection. In fact, differently from operating instructions, which describe how to exploit an artifact, a function description describes what to obtain from an artifact. Clearly, function description is an abstraction over the actual implementation of the artifact: it hides inessential details over the implementation of the service while highlighting key functional (input/output) aspects of it, to be used by agents for artifact selection.

For instance, consider the case of a digital camera. In order to choose a digital camera among all that are available in the market, interested customers usually start by checking the cameras' technical specifications, such as its memory, zoom capabilities, etc.: these

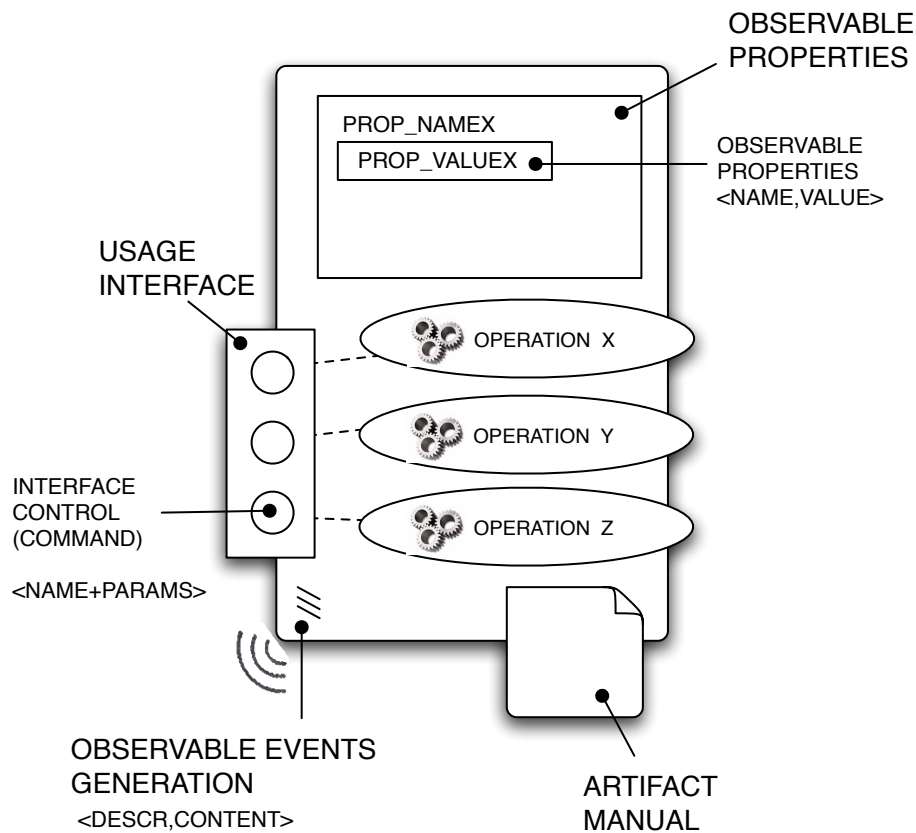


Figure 7.2: A refined model of the artifact

are actually a form of function description—they explain what the camera does. Also, each camera has its own buttons and panels, which must be operated according to the instruction provided by the vendor in the user’s manual. Buttons and panels represent the camera’s usage interface – i.e., how to access it – while the user’s manual describes how to use them to suitably configure the camera resolution, the zoom ratio, etc.—thus representing the operating instruction.

The artifact abstraction leads to a notion of *use* that is the basic kind of relationship among agents and artifacts, besides creation and disposal. Accordingly, a more refined notion of usage interface is defined as the basic set of operations and *observable states* and *events* that an artifact exposes so as to be usable by agents [180]. Informally, an agent can interact with an artifact through its usage interface as follows: an agent executes actions that result in the triggering of some artifact operations, which then leads to the observation of *events* or the evolution of the artifact state (Figure 7.2).

Such an abstraction strictly mimics the way in which humans use their artifacts: a simple example is the coffee machine, whose usage interface includes suitable controls – such as the buttons - and means to make (part of) the machine behaviour observable -

such as displays – and to collect the results produced by the machine—such as the coffee can. It is quite evident by now that, differently from agents, artifacts are not meant to be autonomous or pro-active: they are meant to represent passive entities that are useful if and only if properly (created and) used by agents.

In addition, artifacts typically exhibit further relevant properties, which enhance MAS engineers' but also agents' ability to use them for their own purposes. For instance, it should be possible to *monitor* artifacts as an observable part of the environment, so as to check the development of the activities, track the system history, and evaluate the overall system performance.

In addition, artifacts should exhibit further relevant properties, which enhance MAS engineers' but also agents' ability to use them for their own purposes. For instance, it should be possible to *monitor* artifacts as an observable part of the environment, so as to check the development of the activities, track the system history, and evaluate the overall system performance. Desirable artifact features can then be listed as follows:

Inspectability — The state of an artifact, its content (whatever this means in a specific artifact), its usage interface, operating instructions and function description might be all or partially available to agents through inspectability. Whereas in closed MASs this information could be hard-coded in the agent – the artifact engineer develops the agents as well – in open MASs third-party agents should be able to dynamically join a society and get aware at run-time of the necessary information about the available artifacts. Also, artifacts are often in charge of critical MAS behaviour: being able to inspect a part or the whole of an artifact features and state is likely to be a fundamental capability in order to understand and govern the dynamics and behaviour of a MAS.

Controllability — Controllability is an obvious extension of the inspectability property. The operational behaviour of an artifact should then not be merely inspectable, but also controllable so as to allow engineers (or even intelligent agents) to monitor its proper functioning: it should be possible to stop and restart an artifact working cycle, to trace its inner activity, and to observe and control a step-by-step execution. In principle, this would largely improve the ability of monitoring, analysing and debugging at execution time the operational behaviour of an artifact, and of the associated MAS social activities as well.

Malleability — Also related to inspectability, malleability (also called *forgeability*) is a key-feature in dynamic MAS scenarios, when the behaviour of artifacts could require to be modified dynamically in order to adapt to the changing needs or mutable external conditions of a MAS. Malleability, as the ability to change the artifact behaviour at execution-time, is seemingly a crucial aspect in on-line engineering for MASs, and also a prospective key issue for self-organising MASs.

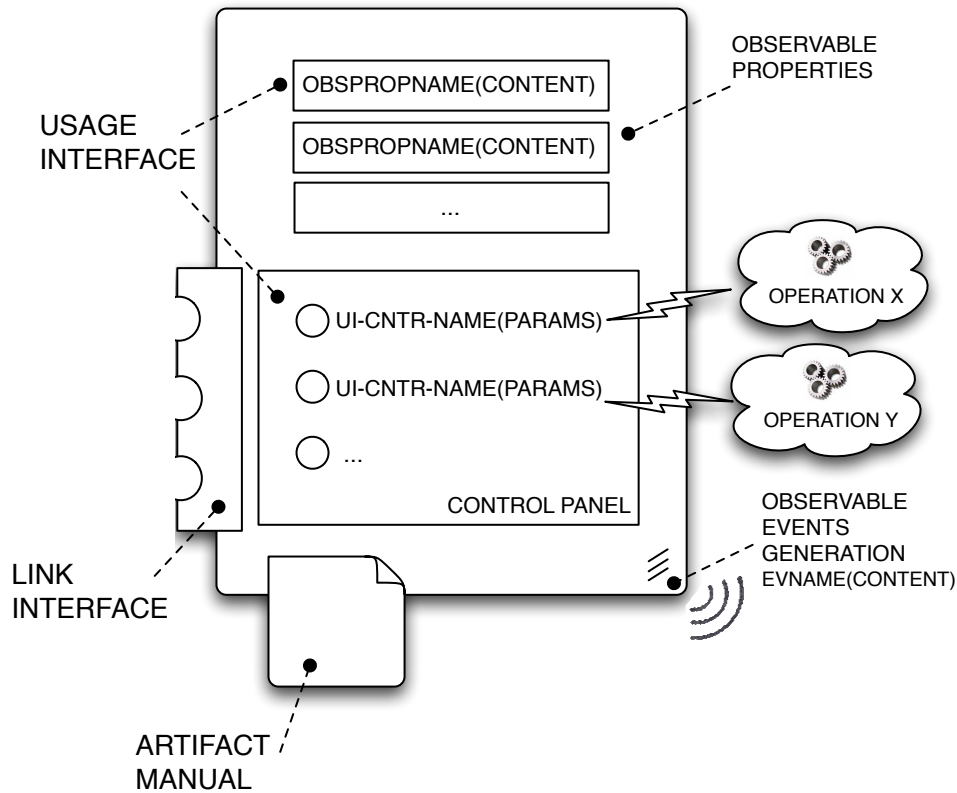


Figure 7.3: Artifact model with linkability

Predictability — Differently from agents – which as autonomous entities have the freedom of behaving erratically, e.g. neglecting messages - usage interface, operating instructions and function description can be used as a contract with an artifact by an agent. In particular, function description can provide precise details of the outcomes of exploiting the artifact, while operating instructions make the behaviour of an artifact predictable for an agent.

Formalisability — The predictability feature can be easily related with formalisability. Due to the precise characterisation that can be given to an artifact behaviour, until reaching e.g. a full operational semantics model – for instance, as developed for coordination artifacts in [153] - it might be feasible to automatically verify the properties and behaviour of the services provided by artifacts, for this is intrinsically easier than services provided by autonomous agents.

Linkability — Artifacts can be used encapsulate and model reusable services in a MAS. To scale up with complexity of an environment, it might be interesting to compose artifacts, e.g. to build a service incrementally on top of another, by making a new artifact realising its service by interacting with an existing artifact. To this end,

artifacts should be able to invoke the operation of another artifact: the reply to that invocation will be transmitted by the receiver through the invocation of another operation in the sender. This leads to refine again the artifact model accounting the linkability feature (Figure 7.3).

Distribution — Differently from an agent, which is typically seen as a point-like abstraction conceptually located to a single node of the network, artifacts can also be distributed. In particular, a single artifact can in principle be used to model a distributed service, accessible from more nodes of the net. Using linkability, a distributed artifact can then be conceived and implemented as a composition of linked, possibly non-distributed artifacts—or vice versa, a number of linked artifacts, scattered through a number of different physical locations could be altogether seen as a single distributed artifact. Altogether, distribution and linkability promote the layering of artifact engineering.

7.2.2 Taxonomy of artifacts

Many sorts of different artifacts populate a MAS, providing agents with a number of different services, embodying a variety of diverse models, technologies and tools, and addressing a wide range of application issues. So, different categorisations could be made. For instance, for coordination artifacts, which entail a form of mediation among the agents using a given artifact and enact some coordination policy, two basic aims can be identified: the *constructive* artifact, as an abstraction aimed at creating and composing social activities; and the *normative* artifact, essential for ruling social activities. This distinction is particularly relevant when dealing with the concept of norm, however for our purposes a different classification seems more useful. The taxonomy of artifacts presented in [151] distinguishes among *individual artifacts*, *social artifacts*, and *environmental artifacts* (Figure 7.4):

Individual artifacts are artifacts exploited by one agent, and mediate between an individual agent and the environment. In general, individual artifacts are not directly affected by the activity of other agents, but can, through linkability, interact with other artifacts in the MAS.

Social artifacts are instead artifacts exploited by more than one agent, and mediate between two or more agents in a MAS. In general, social artifacts typically provide MASs with a service which is in the first place meant to achieve a social goal of the MAS, rather than an individual agent goal.

Environmental artifacts are artifacts that conceptually wrap external resources, and mediate between agents of a MAS and the external resources. In principle, resource artifacts can be conceived as a means to raise external MAS resources up to the

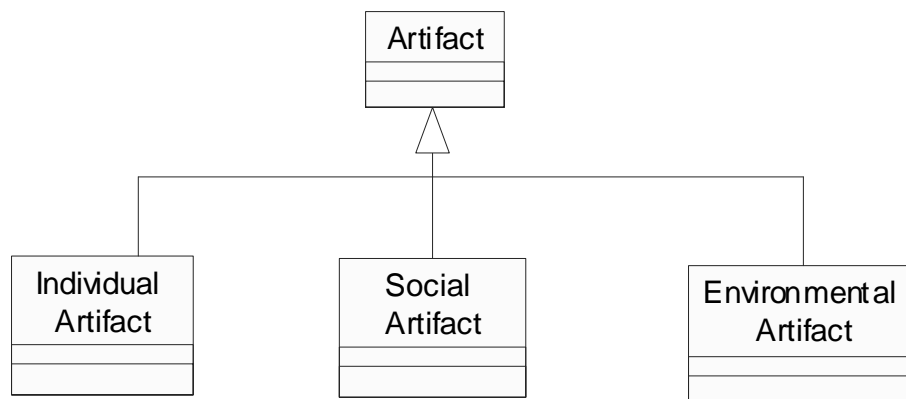


Figure 7.4: Artifacts Taxonomy

agent cognitive level. In fact, they can equip external resources with a usage interface, operating instructions, and a service description, and realise their task by dynamically mapping high-level agent interactions upon lower-level interactions.

In the end, individual, social and resource artifacts can be used as the basis for building the glue keeping agents together in a MAS, and for structuring the environment where agents live and interact: altogether, they can be taken as the conceptual, layered foundation for artifact design in MAS engineering.

7.3 Workspaces

Workspaces can be used to define the *topology* of the working environment. A workspace can be defined as an open set of artifacts and agents creating and using them: artifacts can be dynamically added to or removed from workspaces, agents can dynamically enter (join) or exit workspaces. The same artifact can belong to multiple workspaces. By defining a topology of the environment, workspaces make it possible to structure agents and artifacts organisation and interaction, in particular functioning as scopes for event generation and perception, and artifact access and use. On the one side, a necessary condition for an agent to use an artifact is that it must exist in a workspace where the agent is located. On the other side, events generated by the artifacts of a workspace can be observed only by agents belonging to the same workspace.

Intersection and nesting of workspaces are supported to make it possible to create articulated topologies. In particular, intersection is supported by allowing the same artifacts and agents to belong to different workspaces.

7.4 Summing up

This chapter has introduced the notion of artifact as first-class abstraction for MAS engineering. Along with agents, artifacts constitute the basic building blocks both for MAS analysis and modelling, and for MAS development and actual construction—i.e., real first-class abstractions available to engineers throughout MAS design and development process, down to run-time. Artifacts are objects explicitly designed to provide some function, which guides their use. Typically, artifacts take the form of objects or tools that agents share and use to support their activities, and to achieve their (individual and social) objectives. By adopting a cognitive perspective over systems [153], agents are the entities of a system that are characterised by some goals to be pursued, whereas artifacts are the entities that are not intrinsically characterised by a goal (they are not goal-oriented). Instead, artifacts are characterised by the concept of use, where an agent using an artifact for its own goals implicitly (and temporarily) associates an external goal to the artifact itself. So, agents and artifacts can be assumed as the two fundamental abstractions required to model and shape the structure of MASs: a MAS is made by agents *speaking* with other agents and *using* artifacts in order to achieve their goals.

Part III
Environment

8

AOSE & Environment

Different perspectives exist on the role that environment plays within agent systems: however, it is clear at least that all non-agent elements of a MAS are typically considered to be part of the MAS environment [129]. Current practice in AOSE considers the environment as an implicit part of the MAS that is often dealt with in ad hoc way. Indeed, following [219], the environment should be considered as an *explicit* part of MAS, to be modelled and designed as a *first-class abstraction*. Along this line, MAS environment should be explicitly accounted for in the engineering of MAS, working as a new design dimension for AO methodologies.

While this general perspective on MAS environment is commonly shared in the agent community [220], today AO methodologies actually provide little or even no support for the modelling and design of MAS environment: for instance, a number of them – such as PASSI [35] and INGENIAS [170] – just support environment modelling, while others do not consider MAS environment at all. Even more, it is often difficult to understand how the concept of environment is actually supported by AO methodologies, also in the cases when it is explicitly mentioned. In the PASSI meta-model, for instance, the modelling of environment – from the agent viewpoint – is somehow “hidden” in the ontology description, and is scarcely highlighted; then, typical environmental elements like physical resources are represented as mere agent-related entities and are strangely not related to the ontology description.

This chapter reviews some of the most significant AO methodologies according to their approach to the engineering of MAS environment. First of all the methodologies are classified in three different groups, and explain the rationale for this choice (Section 8.1). Then an overview of the methodologies is presented according to the previous classification. In particular, Section 8.2 presents the so-called strong-env methodologies, Section 8.3 illustrates weak-env methodologies, whereas no-env methodologies are presented in Section 8.4. A summary is reported in Section 8.5

8.1 Classification of AO Methodologies

As mentioned above, the goal of this Chapter is to study AO methodologies (Chapter 4) in order to understand how they deal with the engineering of MAS environment. In particular, the focus is on how methodologies model and design the environment, and on what kind of environment abstractions and topology abstractions (see Chapter 2, Section 2.3) they provide to MAS engineers.

Each of AO methodologies has obviously its own strengths and drawbacks, in general. For example, Tropos is a good methodology for the requirements capturing and analysis but it does not consider environment engineering, while ADELFE is good in the environment engineering but does not say enough on requirements capturing. However, the aim of this classification is not to present a classical, general-purpose survey and comparison of AO methodologies. Rather, the viewpoint is stuck to engineering MAS environments, and to try to provide the reader with a clear overview of how existing AO methodologies deal with them. So, this classification should not be taken as a statement of why a certain methodology is better or worse than another, but just as a measure of how much it supports MAS engineers in dealing with the environment.

Accordingly, though AO methodologies are quite heterogeneous, it is obviously useful to classify them according to the extent by which they tackle environment in MAS:

strong-env *Strong environment viewpoint*: this kind of methodology allows MAS engineers to both model and *design* the environment at every stage of the methodology.

weak-env *Weak environment viewpoint*: methodologies belonging to this category take into account only the *modelling* of the environment.

no-env *No environment viewpoint*: in short, these methodologies do not consider environment at all, at least explicitly.

The difference between strong-env and weak-env methodologies is crucial here: in the former case, MAS engineers can fully design the environment through suitable dedicated abstractions provided by the methodology; in the latter case, environment structure and behaviour are taken as given a priori, and MAS engineers can only model them. In other terms, weak-env methodologies mostly deal with a notion of MAS environment as the external environment, while strong-env methodologies typically recognise the existence of an agent environment within a MAS that can be suitably modelled and designed by MAS engineers.

8.2 Strong-Env Methodologies

Methodologies in the *strong-env* category promote MAS environment as a *first-class abstraction* [219] in the modelling and design of MAS from the early phases of the develop-

ment process. In this category we find ADELFE and OperA+Environment ¹.

8.2.1 ADELFE

In ADELFE [9, 173], the environment is studied since the WorkDefinition 2 (WD2, Final Requirements) stage, where MAS environment is characterised in terms of the entities (either passive or active) that interact with the MAS. An *active entity* can behave autonomously and is able to dynamically interact with the system, instead a *passive entity* can be considered as a resource exploited by the system, which cannot change in an autonomous way (these will later become *objects* of the environment). In the subsequent stage, MAS context is studied through the interactions between the entities and the system. Finally, the environment is described in the terms of *accessibility*, *continuity*, *determinism* and *dynamism*.

In the WD3 (Analysis) stage, active entities are split in two sets: *cooperative* entities (that will become agents), and the *active entities* that autonomously evolve without having a goal. The latter are autonomous resource or active objects, and will remain simple objects in the environment. Afterwards, in WD3 interactions between entities (passive and active) are designed by means of standard UML [143] sequences or collaboration diagrams, as advised by RUP. Then, the design of “environment abstractions” (passive and active entities) is done according to the RUP guidelines, determining packages, classes and using traditional design patterns. This implicitly defines the behaviour of the environment abstractions. Also, the topology of environment is implicitly defined by means of an agent internal module called “representation module”, which enables agents to create their own representation of the environment they perceive, and also by means of the study of MAS context in WD2.

8.2.2 OperA+Environment

The OperA+Environment approach [40] adopts the analysis from OperA [52], a methodology that uses organisation structures and provides for open agent systems. The authors have refined OperA with the introduction of both the environment model and the design phase.

The environment model in the analysis phase specifies the *resources* that are available for the agents, like databases, etc. The model also specifies the available *services* – like white or yellow pages, but also like the Mathematica package to support calculations of any sort – as well as the way the system can interact with the environment.

In the design phase the Infrastructure Model makes use of the design models from two existing methodologies: **SODA** [144] and Gaia [225]. This model consists of a resource model, a service model, and a model of coordination facilities. Each resource model

¹In this Chapter the **SODA** methodology is not considered. **SODA** will be extensively presented in Chapters 13 and 14

	Environment Abstractions	Behaviour of Abstractions	Topology Abstractions
ADELFE	<i>passive and active objects</i>	<i>implicitly designed</i>	<i>implicitly defined</i>
OperA+ Environment	<i>resources, services and coordination facilities</i>	<i>explicitly designed</i>	<i>not supported</i>

Figure 8.1: Strong environment viewpoint in AO methodologies

contains a specification of the resource (e.g. document, database, or library) in terms of various qualities and quantities, access permissions, and admissible actions. The service model defines which services (e.g. any activity that processes information) can be delivered to a certain service request. The services can be defined as abstract operations in terms of input and output functionality. For each service, the model specifies the permissions needed to use the service by an agent playing a certain role. It also specifies the quality of service, such as the maximal delay in providing the service, the format of messages it can process, and the protocols it can support. Finally, *coordination facilities* are mechanisms that can be used by the agents to coordinate their activities. Examples are synchronisation, tuple spaces, subscribe-notify design pattern, or various types of protocols.

8.2.3 Strong-env Methodologies at a Glance

Table 8.1 summarises how the methodologies support the key elements of strong environment viewpoint, and classifies them along three dimensions: environment abstractions, behaviour of abstractions, and topology abstractions.

8.3 Weak-Env Methodologies

Methodologies in this category take into account the environment as an entity which is given a priori, which MAS engineers can only model. Environment modelling may occur at different stages of such methodologies. In this category Gaia, PASSI, MESSAGE, INGENIAS, Prometheus, and ROADMAP can be found.

8.3.1 Gaia

In Gaia [228, 229], MAS environment is studied from the analysis phase. In particular the *environmental model* is intended to make explicit the features of the environment in which the MAS will be immersed. The identification and modelling of the environment involves determining all the entities and resources that the MAS can exploit. Gaia suggests

treating the environment in terms of *abstract computational resources* (such as variables or tuples) made available to agents for sensing, affecting or consuming. Following such an identification, the environmental model can be viewed as a list of resources characterised by the type of the actions that the agents can perform on it. In the *preliminary role model*, roles are related to environment by means of *permissions*. Permissions discipline how roles can access environmental resources and possibly change or consume them. In order to represent permissions, Gaia adopts the same notation used for environmental resources. However, the attributes associated with resources no longer represent what can be done with such resources (i.e., reading, writing, or consuming) from an environmental perspective, but rather what the agents playing the role are allowed to do (or not to do) to accomplish the role goal(s).

In the architectural design, the preliminary role model is completed by the addition of roles generated by the architectural choice. For each new role, permissions associated to environmental resources are defined. In the detailed design, agents are associated with roles, so that agents are related to environmental resources.

8.3.2 PASSI

In PASSI [35, 38], the environment is studied from the Agent Society Model, in particular in the Ontology Description Phase. In the Domain Ontology Description, agents know the environment through the abstractions of *Concepts* (categories, entities of the domain), *Predicates* (describing the state of the instance of concepts that actually occur in the environment), and *Actions* (performed in the domain), along with their mutual relationship. In the Communication Ontology Description the agent interactions are represented: in each communication, it is important to introduce the proper data structures (selected from elements in Domain Ontology Description) within each agent in order to store the exchanged data. In the PASSI meta-model the concept of *Resource* also appears, which represents a tangible entity of the environment with which agents can interact, and which is a part of agents' environment awareness. This concept only relates with agents: no relation exists with ontology elements.

In the Role Description Phase PASSI introduces a relationship called *Resource Dependency*: a role depends on another for the availability of an entity (indicated by a *resource name*)—however, it is unclear from the available documentation whether this resource is the same that appears in the meta-model.

8.3.3 MESSAGE

MESSAGE [20, 69] has adopted the Rational Unified Process (RUP) as a generic software engineering project lifecycle framework. MESSAGE adopts the abstraction of Resource as a Concrete Element in the system. Resource is used to represent non-autonomous entities such as databases or external programs used by Agents. Standard object-oriented

concepts are adequate for modelling Resources.

Then, in the analysis phase, several views take into account the concept of resource. In particular the Organisation view shows Concrete Entities (Agents, Organisations, Roles, Resources) in the system and its environment, as well as coarse-grained relationships between them (aggregation, power, and acquaintance relationships). An acquaintance relationship indicates the existence of at least one Interaction involving the entities concerned. In addition, the Domain view shows the domain specific concepts and relations that are relevant for the system under development. The Agent/Role view specifies for each agent/role what resources it controls.

The purpose of the design phase is to define computational entities that represent the MAS appearing at the analysis level. Analysis entities are thus translated into subsystem, interface, classes, operation signatures, algorithms, objects, object diagrams, and other computational concepts.

8.3.4 INGENIAS

INGENIAS [96, 170] provides a notation for modelling MAS, and a well-defined collection of activities to guide the development process of MAS from tasks to code generation. INGENIAS uses the concept of *viewpoint* as MESSAGE. Each viewpoint in INGENIAS is constructed following two sets of activities structured into activity diagrams: one set aims at elaborating the view at the analysis level, whereas the other focusses on the design. In particular the *Environment Viewpoint* defines the entities with which the MAS interacts, which could be *Resources* (such as CPU, File Descriptors or memory), *Other Agents* (from existing organisations), and *Applications* (expressing the perception and action of the agents, producing the events that can be observed).

In the Development Process INGENIAS suggests to identify in first place all the software systems (usually non-agent based) that will coexist with the MAS. By using the environmental model, it is possible to consider dependencies – in terms of agent perceptions and actions – with legacy and proprietary systems from the beginning.

8.3.5 Prometheus

Prometheus [163, 164] is intended to be a practical methodology: it aims at being complete and detailed, and to be usable by industrial software developer. In Prometheus the environment is modelled since the System Specification phase. In particular, the environment is defined by describing the percepts available to the system, the actions that it will be able to perform, and any external data that are available as well as any external bodies of code.

In the Architectural Design phase the environment appears in the design of the overall structure of the systems: the overview diagram captures the agent types, the boundaries of the system, and its interface in terms of actions and perceptions, but also in terms of

data and code that are external to the system. In the Detailed Design phase the overview diagram is used to develop internals of agents and interaction protocols.

8.3.6 ROADMAP

The ROADMAP [103] methodology extends the first version of Gaia with several features: support for requirements gathering, explicit models to describe the domain knowledge and the execution environment, levels of abstraction during the analysis phase, to allow iterative decomposition of the system, explicit models and representations of social aspects and individual agent characteristics, from the analysis phase to the final implementation, runtime reflection, modelling mechanisms to reason and change the social aspects and individual agent characteristics at runtime. The environment model is proposed to provide a holistic description of the system environment and provides both the environment abstractions and the modelling of the topology. By formally describing the environment, the authors create a knowledge foundation on which environment changes are handled consistently. The environment model is derived from the use-case model. The model contains a tree hierarchy of *zones* in the environment, and a set of *zone schema* to describe each zone in the hierarchy. A zone schema includes a text description of the zone, and the following attributes: *static objects*, *objects*, *constraints*, *sources of uncertainty* and assumptions made about the zone. Static objects are entities in the environment whose existences are known to agents, with which agents do not interact explicitly. Objects are similar entities with which agents interact. Sources of uncertainty in the environment are identified and analysed. The zone hierarchy uses OO-like inheritance and aggregation to relate zones and various objects inside zones.

8.3.7 Weak-env Methodologies at a Glance

Table 8.2 summarises the environment-related abstractions adopted by weak-env methodologies. This table, unlike Table 8.1, only presents environment and topology abstractions, while the behaviour of the abstractions is omitted because here MAS environment is only modelled and not designed.

8.4 No-Env Methodologies

Methodologies in this category do not deal with the concept of MAS environment. Tropos and MaSE can be taken as representatives of this category.

8.4.1 Tropos

Tropos [13, 77] is a requirements-driven methodology and adopts the abstractions offered by *i**, a modelling framework proposing concepts such as actor (actors can be agents,

	Environment Abstractions	Topology Abstractions
Gaia	<i>abstract computational resources</i>	<i>not supported</i>
PASSI	<i>resource, concepts, predicates and actions</i>	<i>not supported</i>
MESSAGE	<i>resources</i>	<i>not supported</i>
INGENIAS	<i>resources, other agents and applications</i>	<i>not supported</i>
Prometheus	<i>actions, perceptions, external data and code</i>	<i>not supported</i>
ROADMAP	<i>static objects and objects</i>	<i>zones, zones schema, constraints</i>

Figure 8.2: Weak Environment Viewpoint in AO methodologies

positions or roles), as well as social dependencies among actors, including goal, softgoal, task and resource dependencies. These concepts are used in all software development phases of Tropos, from the early requirements analysis down to the actual implementation. Resources are always involved in dependencies among actors. A resource is not an abstraction that models the environment, instead it could be thought of as an item for knowledge exchange among actors: more precisely, a resource represents a physical or an informational entity that one actor may want and another could deliver.

8.4.2 MaSE

MaSE [44, 224] guides a designer through the software lifecycle from a prose specification to an implemented agent system. MaSE is independent of a particular MAS architecture, agent architecture, programming language, or message-passing system. The abstractions used by MaSE are goals, tasks, roles and agents. Each goal is associated to the role that achieves it. Any mention of separate machines or other forms of distribution requires one role for each “side” of the distributed relationship. Interfacing with an external source is the same. One role may interface with the source while another may be required to bridge the gap back to the system. This is also true for any database, file interface, or user interface in the system. A user interface implies a role by itself, and should be separated from other roles as if it were a distinct data source. Then role are assigned to agents—so in the end everything is an agent in MaSE.

8.4.3 No-env Methodologies at a Glance

In this subsection two different examples of methodologies that do not explicitly consider the environment, yet have been briefly reviewed. Typically in this kind of methodology

the environment is not totally forgotten, instead it is typically represented by means of agents—so, not as a first-class entity. A common viewpoint is that as far as AOSE is concerned using agents for modelling everything means somehow to pervert the nature of the agent abstraction itself [219]. For instance, the agent abstraction is not the most suitable one for modelling resources, for which crucial agent concepts like autonomy and proactiveness simply do not apply. Instead, relying on notions like environment and topology abstraction is a key step to tackle the peculiarities of environment in AO methodologies.

8.5 Summing up

The key role of environment as a first-class abstraction in the engineering of MAS is today generally acknowledged in the MAS community.

However, this chapter has highlighted that in the AOSE field a small number of methodologies actually deal with environment as a first-class abstraction, while some others provide MAS engineers with only one model of the environment, and a few others do not consider environment as a first-class abstraction at all, yet. Furthermore, even in the case of methodologies actually modelling MAS environment, such a feature is often somehow hidden or not-well documented. The proposed survey has tried to understand in depth how each methodology handles MAS environment, and which sorts of environment abstractions it adopts. The results are then a classification of methodologies in three different categories: strong-environment viewpoint (methodologies that consider environment as a first-class abstraction for MAS engineering), weak-environment viewpoint (methodologies that only model environment) and no-environment viewpoint (methodologies that do not consider environment), and an overview of how the environment is managed by the methodologies presented according to the previous classification.

9

Environment in AO Methodologies

This chapter presents a viable approach to introduce the notion of environment within existing AO methodologies [129]. This will be done by extending AO methodologies with ad hoc method fragments – e.g. portions of the development process – [63] that allow methodologies to support modelling and designing of the environment without re-drawing the whole methodology. First of all, an identification of the environment-related abstraction which will be adopted in the approach should be done (Section 9.1). This abstraction should be general enough to fit most methodologies. Among the available alternatives, the abstractions introduced by the A&A meta-model (A&A) [178] – namely, *artifacts* and *workspaces* – seem good candidates. These abstraction are adopted by the new version of the **SODA** methodology (Chapter 14). While this choice is obviously not the only one possible (Section 9.1), the remainder of this chapter presents how AO methodologies can be extended with artifacts and workspaces to generally deal with the engineering of environment in MAS.

In the discussion that follows, no-env and weak-env methodologies are kept distinct: whereas neither has a complete handling of environment, each has a different approach toward it. So, the indentification of the environment and topology abstraction is reported in Section 9.1, while the next sections first show how to introduce environment in a no-env methodology so as to make it a weak-env one (Section 9.2), then discuss how to transform a weak-env methodology to a strong-env one (Section 9.3). Related work and the summary follows in Section 9.4 and Section 9.5.

9.1 Environment and Topology Abstractions

When dealing with the introduction of the environment in a methodology, the first issue to consider is the choice of the most suitable environment abstraction. The notion of environment abstraction as introduced in Section 2.3 is a high-level notion: while it is a good way for discussing the environment in general, it might be practical to exploit a more concrete notion at the methodological level. In fact, since it is an abstraction over the real entities used by infrastructures, there is no clear definition of its concrete structure, and of the concrete interaction modality it promotes.

In order to choose a suitable environment abstraction, it might be useful to take inspiration from the abstractions already used both by existing infrastructures for MAS and by AO methodologies. Entities provided by infrastructures seem to be too specific because they are typically conceived for a particular application domain, so they are not easy to adapt to general modelling—as required by a general-purpose methodology. On the other hand, AO methodologies – especially those in the weak-env group – use very different notions of environment abstractions, belonging to different cognitive levels: from lower-level ones such as *external data and code*, *ontologies* and *objects*, to higher-level ones such as *resources*, *abstract computational resources*, and *artifacts*.

This problem, associated with the typical openness requirements for MAS, leads to seeing the agent interaction space as spanning over different levels. There, agents cannot interact with the other components (agents and environment abstractions) of the MAS in a uniform way: they communicate with each other via high-level languages, while often using lower-level languages – one for each different kind of environment abstraction – as a way to interact with the environment. From the general software engineering point of view this situation is not acceptable because it produces an unnecessary complication in the engineering of both the agent internal structure and the interaction protocols. The proliferation of environment abstractions seems to be more a drawback than a strength because it does prevent a clear and common view of environment engineering. In addition, it also seems an obstacle in the path toward the creation of a general-purpose AO methodology. The first and obvious choice among the environment abstractions could be the adoption of the object as a unique building block in the environment engineering because the object is a well-known and widely used technology and design paradigm. However, objects seem to be not so suitable in the context of environment engineering because they provide a too low level of abstraction than agents and they do not provide natively all the mechanisms for supporting the agents in the discovering and the selection of the most suitable service among different choices. These mechanisms are fundamental in order to promote the MAS openness and the agent’s mobility. In addition the objects are not able to wrap several of the other environment abstractions since many of them belong to an higher level of abstraction.

Seemingly, one of the most promising environment abstraction is the notion of *artifact*—as highlighted in Subsection 2.3.1. In fact, artifacts are generally defined as both conceptual and runtime entities that mediate agent activities [215], providing some kind of function, or service, that agents can fruitfully exploit to achieve their individual or social objectives. In particular, in the Agents & Artifacts (A&A) meta-model [146, 153] recently proposed for MAS engineering, artifacts – along with agents – are adopted as the basic building blocks to engineer complex software systems. Artifacts are the basic abstractions to represent passive, function-oriented building blocks, which are constructed and used by agents, either individually or cooperatively, during their working activities. Many sorts of artifact can populate a MAS: in particular, artifacts are used to mediate between individual agents and the MAS (individual artifacts), to build up agent societies

(social artifacts), and to mediate between a MAS and external resources (environmental artifacts) [152].

The second issue concerns the choice of the most suitable abstraction for modelling MAS topology. As for the environment abstraction, there is a need for a general-purpose abstraction able to capture different deployment contexts. However, different from the environment abstraction case, the topology abstractions provided by both MAS infrastructures and AO methodologies are few, and the level of abstraction provided by them is very low. Here, *workspaces* – introduced by **CARTAgO** in the A&A meta-model, and also adopted by **SODA** – seem a suitable choice since they are general enough for abstracting any sort of topology, and are also closely tied with the concept of artifact.

9.2 From No-Env to Weak-Env Methodologies

The introduction of the environment in a methodology implies a significant modification of the first phases of the methodology, and as a consequence a partial modification of the other phases.

9.2.1 Requirement Specification

The first step is a thorough analysis of the requirements specification that concerns environment, with the following goals—see top of Figure 9.1:

- (i) Detecting the presence of legacy systems that the MAS should interact with. Legacy systems are computational systems that have already been installed and are working, and which the designer of a new MAS should exploit without the possibility of changing them in a significant way. These systems obviously will become part of MAS environment, since they often provide several services to the MAS under development.
- (ii) Recognising those requirements concerning function-oriented and passive entities, which should not become agents for they have neither goals nor autonomy. However in some cases it is not so easy to understand when an entity is not an agent, since the concept of “function-orientation” is not always tied with the concept of passivity.
So, recognising when a requirement deals with non-agent entities is a non-trivial job as well, because a requirement typically deals with both several (future) agents and non-agent entities, and such entities are closely tied. As a remark, here such function-oriented entities are supposed to be already implemented, or anyway provided by someone, so they are only modelled. The design of these entities is one of the topics of next subsection.
- (iii) Individuating topological constraints over the structure of the deployment context. Typically in the early phase of requirements analysis, such constraints represent

physical ties over the environment, but sometimes some constraints could come from requirements.

It would be useful to keep track of the relations between requirements and legacy systems / function-oriented entities in order to create those links that will become the interaction protocols between agents and artifacts. The protocols will be properly designed later in the design phase. In addition, also the relations between requirements and constraints could be examined in order to facilitate the definition of workspaces.

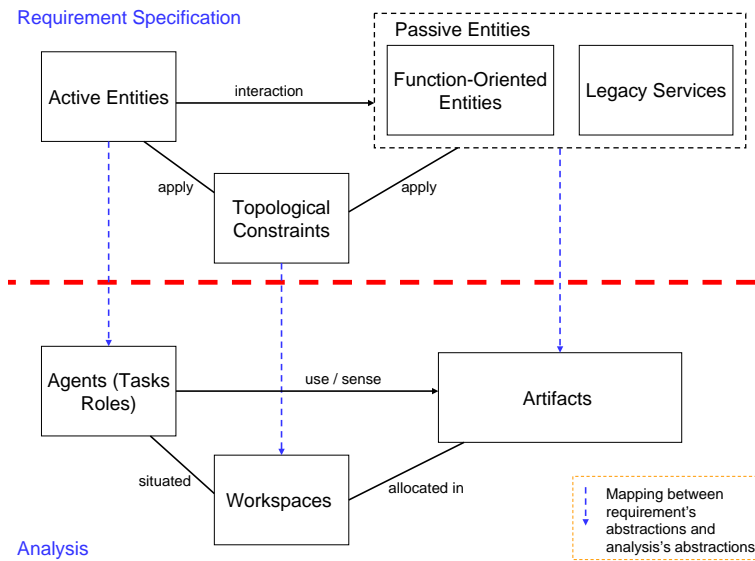


Figure 9.1: Relations between requirement abstractions (top) and analysis abstractions (bottom)

9.2.2 Analysis

In the analysis phase, a model of environment composed of artifacts and workspaces is built: both the legacy systems and the function-oriented entities are modelled as artifacts (see Figure 9.1). According to [178], this means specifying for each of them the *usage interface* (the operations provided by the artifact), the *operating instructions* (how to access the artifact), and the *function description* (what services are provided by the artifact). Then the relationships previously identified, between requirements and the function-oriented entities / legacy systems, are refined in more concrete relations between the system abstract entities (adopted in the other original models of the methodology) and artifacts. Finally, the topological constraints generate the workspaces for structuring the environment. Sometimes, intersection and nesting of workspaces are here necessary

in order to create articulated topologies coming from very complex constraints. Moreover, artifacts are allocated to the most suitable workspace(s) according to the specific topological constraints.

9.2.3 Design

In the design phase(s) of the methodology, the environment model does not change, only the interaction protocols are designed starting from the relations sketched out in the analysis phase. If the methodology schedules other phases such as fast prototyping or code generation, these should be adapted in order to support the generation of the new interaction protocols between agents and the entities in the environment. As interactions with the environment are uniformly seen as interactions between agents and artifacts, all the above phases should apparently be no more complex than the design of MAS relying on standard agent communication protocols. In addition, if the methodology to be extended presents deployment models, these should be changed so as to support the structure of environment coming from workspaces. Simply, that model will need to be extended so that it can support the new “workspace” abstraction and the relations between workspaces.

9.2.4 An Example: Tropos from No-Env to Weak-Env

As an example methodology to ground our discussion we consider Tropos, which is a methodology where requirements are well structured and documented. Tropos could represent an exception in the proposed method, since it adopts a particular framework for the requirement analysis and needs a careful investigation. Requirements analysis in Tropos is split in two main phases: Early Requirements and Late Requirements analysis [13]. More precisely, during the first phase, the requirements engineer identifies the domain *stakeholders* and models them as social actors, who depend on one another for *goals* to be achieved, plans to be performed, and resources – remember that this concept does not represent an environment abstraction – to be provided.

Following our method, first of all in the Early Requirement analysis the identification of legacy system is needed. So, while the requirements engineer identifies the domain stakeholders, he/she should also analyse the system specification in order to discover those legacy systems the new MAS will interact with. In particular the requirements engineer should identify the relationships between the legacy systems and stakeholders (if any exist). If legacy systems are related with actors’ goals, then such relations need to be traced. In addition, the requirements engineer should also recognise whether legacy systems are subject to some particular constraints over the environment structure. Legacy systems are then modelled as artifacts, and the topological constraints generate the workspaces. All the relationships among the entities should be traced.

In the Late Requirements analysis, the conceptual model previously identified is extended including a new actor representing the system, and a number of dependencies with other actors. These dependencies define all the functional and non-functional requirements of the system-to-be. Following our approach, this is the right stage for individuating function-oriented and passive entities, and the other topological constraints. In particular, the requirements engineer should check carefully both the functional and non-functional requirements searching those function-oriented entities that either provide services to actors or support them in the achievement of their goals. Topological constraints typically come from non-functional requirements, so they should be studied more deeply in order to recognise all the constraints over the environment structure. Subsequently, these entities become respectively artifacts and workspaces, and the links among artifacts and goal / actor should be traced.

The Architectural Design and the Detailed Design phases in Tropos focus on the system specification, according to the requirements resulting from the above phases. Architectural Design defines the system global architecture in terms of sub-systems, interconnected through data and control flows. Sub-systems are represented as actors, and data/control interconnections are represented as dependencies. Following our approach, in this phase the interaction protocols among actors/agents and artifacts should be refined according to the specific architectural structure adopted. In particular the choice of a specific architecture could lead to refining the actor model and as a consequence new and/or more refined relations among actors and artifacts could be discovered. In a similar way, the choice of the architecture could lead to identifying new topological constraints over the environment and as a consequence a new refined workspaces structure could be outlined.

The architectural design provides also a mapping of the system actors onto a set of software agents, each characterised by specific capabilities. The Detailed Design phase aims at specifying agent capabilities and interactions. At this point, usually, the implementation platform has already been chosen: this can be taken into account so as to perform a detailed design that will map directly onto the code. Following our approach, in this phase the interaction protocols among agents and artifacts should be designed.

Finally the Implementation activity in Tropos follows step by step, in a natural way, the detailed design specification on the basis of the established mapping between the implementation platform constructs and the detailed design notions. Following our approach in this activity it is necessary to implement the interaction protocols among artifacts and agents, and to organise the environment structure according to the workspaces structure.

In all, then, our method makes the introduction of the environment in Tropos not so difficult. Similar considerations could be done also for MaSE, and possibly for any other no-env AO methodology.

9.3 From Weak-Env to Strong-Env Methodologies

Weak-env methodologies already deal with some kinds of environment abstractions. In order to simplify their treatment, and also to make methodologies homogeneous, as a first step we suggest the artifact-based wrapping of the different existing environment abstractions, as explained in Section 9.1. Then, in order to complete the analysis phase it is possible to go back to the requirements, and acting in a similar way as discussed in the previous subsection. However, the identification of legacy systems is not necessary here, since they have yet been modelled according to the original environment model.

9.3.1 Requirement Specification

The recognition of function-oriented entities – as highlighted in point *(ii)* of requirement specification in the previous subsection – here it is not simply aimed at modelling entities already implemented in the environment, but also at discovering new function-oriented entities that are necessary to the MAS but have not already been identified. Obviously, the relations of these new entities with other abstract entities should be soon determined: first, by outlining the links between function-oriented entities and other requirements, then sketching out the more refined relations between function-oriented entities and the abstract entities adopted by the methodology—see top of Figure 9.1. Subsequently, if the methodology does not deal with topological aspects, these should be introduced by finding out the topological constraints from the requirements.

9.3.2 Analysis

In this phase the function-oriented entities are modelled as artifacts. In addition the discovery of new artifacts is also possible during this phase, when the requirements are well structured and modelled by means of appropriate abstract entities such as roles, tasks, goals and so on. In this case, it is easier to understand whether an active entity needs some kinds of services that are not present yet in order to achieve one or more objectives.

Finally, workspaces are generated from topological constraints, and allocating artifacts and agents inside them. As for artifacts, during the analysis phase new workspaces could be discovered so as to better structure the environment.

9.3.3 Design

In the design phase, the discovery of new artifacts is also possible. In this phase the choices about how to structure the system are made, and the detection of new services is not so difficult. As an example, let us suppose we use a group of roles/agents for the achievement of a particularly complex task/goal. Borrowing a group of roles/agents as

responsible for one task/goal leads to manage the problem of how to coordinate these entities so as to achieve such task/goal. In this case, the solution could be the adoption of a suitable coordination artifact [153, 216] for managing coordination among roles. This artifact could not be discovered in the previous phases because it comes from a specific design choice—whereas other choices could lead to other solutions, as for example the design of complex coordination protocols among agents/roles without using artifacts.

Then, first of all the interaction protocols between agents and artifacts are designed starting from the relations sketched out in the analysis phase; moreover, the required artifacts are designed. The “external behaviour” of an artifact is defined by means of interaction protocols in which it is involved. The internal design could be made by following a traditional software engineering design, as for example the object oriented design, since the internal machinery of artifacts pretty much resembles that of objects—both being passive entities with no proactive behaviour. Note that, though specific guidelines for the internal artifact design have not been defined yet, some guidelines about the design of operating instructions are presented in [216].

If the methodology supports other phases such as fast prototyping or code generation, these should be adapted in order to support the generation of the new interaction protocols between the agents and the artifacts, as well as the code generation for the artifacts and for the workspaces. In addition, if the methodology presents deployment models, these should be changed for supporting the structure of environment as it comes from workspaces.

9.3.4 An Example: Gaia from Weak-Env to Strong-Env

For the sake of concreteness, a sketch of how to transform Gaia in a strong environment methodology is reported. In Gaia the environmental model can be viewed as a list of resources, each denoted by a symbolic name, characterised by the type of actions that the agents can perform on it, and possibly associated with additional textual comments and descriptions [229]. Following our approach, first of all the resources already belonging to the environmental model should be wrapped by means of artifacts; then the requirements should be investigated looking for other function-oriented entities and for all the existing legacy systems. Both should become artifacts. In addition the requirements engineer should recognise whether legacy systems are subject to some particular topological constraints over the environment structure. Such constraints eventually generate the workspaces. All the relationships among the roles identified in the Preliminary Role Model and the artifacts should be traced during the construction of the Preliminary Interaction Model. Finally during the Analysis phase topological aspects could be deduced also from organisational rules, generating workspaces and allocating roles and artifacts inside them.

In the architectural design, the choice of the best system architecture is made, leading to the refinement of the role model – adding new specific roles coming from the choice of the architecture – and the interaction model. Following our approach, in this phase

the choice of the particular architectural structure typically leads to the discovery of new artifacts needed to support the new role activities. In addition, the architecture choice leads to refine the interaction model and as a consequence new and/or more refined interaction protocols among roles and artifacts could be discovered. In a similar way the choice of the architecture could lead to identifying new topological constraints over the environment and as a consequence a new refined workspaces structure could be delineated.

Finally in Gaia, in the detailed design stage, roles are assigned to agents, and artifacts should be internally designed according to a traditional software engineering design. In addition the interaction protocols among agents and artifacts should be designed.

In the end, this example apparently suggests that there is a general way for artifacts and workspaces to be adopted inside a weak-env methodology so as to make it a strong-env one.

9.4 Related Work

Literature dealing with both environment and agent-oriented methodologies is not abundant. Work of that sort is usually more focussed on sociality features, so they typically present some comparison among methodologies (one example is in [87]) and simply point out whether a methodology supports the concept of environment or not. As an example in [203] a good framework for the evaluation of the methodologies is presented, accounting for a large number of agent features (organisations, roles, beliefs, desires and so on) as well as some criteria for the evaluation of the development process: however, environment is not considered at all.

At the same time, so little research has been devoted to the issue of how to introduce the notion of environment in a methodology—an AO methodology, in particular. To the best of our knowledge, there is only one work [45] investigating how to introduce an environmental model in a methodology. The methodology considered there is O-MaSE (Organisation-Based Multiagent System Engineering [42]), an evolution of MaSE dealing with the design of organisational MAS. The authors describe an approach to the modelling of MAS interactions with its environment: the key concepts in their approach are capabilities and the environment model. In particular, the environment is modelled as a set of objects/agents, and a set of relations between such objects/agents. Through a set of capabilities belonging to agents, agents have access to a set of operations that they may perform upon environment objects, whose effect is governed by environmental laws. This work is obviously very interesting in this context, and in the conclusions the authors argue about a possible integration of the concepts of their AEI (Agent-Environment Interaction) Model into existing methodologies. However, they do not specify *how* to introduce their key concepts into other methodologies, so their approach turns to be too strict in scope. Also, even in principle, such a process looks not so easy: in fact, the meta-model of AEI is largely tailored upon the MaSE meta-model, and some of the elements implicitly require

the MaSE specific agent model. As a result, the introduction of environment according to AEI seems to be potentially invasive, since it would involve the creation of a new model for the environment, and also affect the agent model: in fact, adopting the AEI meta-model would tie agents with the concept of “capabilities”, thus requiring a substantial change to the original agent model of the methodology to be extended. Even more, AEI does not consider topology at all.

Another interesting work is the methodology proposed by Simonin and Gechter [198] that establishes the link between the representation of the problem, expressed as environmental constraints, and agent behaviours, which are regulation items of the environmental perturbations. The environment is modelled by means of the definition of its structure (the topology) and the laws that govern its dynamics. Then no environment abstractions are considered by the methodology for representing the environment, so it is not clear how and where the environment laws are enforced.

Finally, also the work on methodology fragmentation conducted by IEEE-FIPA Methodology Technical Committee [63] is very interesting in our environment-AO perspective. In fact, the Committee has developed a method for assembling pieces out of the methodology processes starting from a meta-model of methodologies. An interesting fragment [1] is provided by the ADELFE methodology: there, in fact, the environment model is extracted from the methodology, and a new fragment is create. However, the fragment uses the specific environment abstractions adopted by ADELFE, which appear not general enough to be widely applied to other methodologies. In addition the fragment is not yet well documented, and it is then not so clear what is actually needed for merging the fragment within a methodology.

9.5 Summing up

This chapter has proposed a possible method for introducing the treatment of environment in no-env methodologies—thus transforming them in weak-env methodologies; afterwards, it has shown how to transform weak-env methodologies into strong-env methodologies. To this end, the chapter has shown how to exploit artifacts as general-purpose environment abstractions, and workspaces as abstractions for modelling topologies. These abstractions have already been fruitfully used in a strong-env methodology like **SODA** [144].

The adption of this method seems easy and immediated as the above examples (Subsection 9.2.4 and Subsection 9.3.4) have shown. In addition this method is potentially applicable to every AO methodology, since it does not require many changes: the only modifications required concern the design of the interaction protocols that tie the agents to the environmental entities, and obviously also the insertion of the proposed environmental models into all the phases of the methodology to be extended.

10

AOSE & Infrastructures

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 [148]. A large class of distributed systems need not to be built from scratch but can exploit infrastructures to resolve heterogeneity and distribution of the system components. The software engineering challenges lie in devising methodologies, notations and tools for distributed system construction that systematically build and exploit what infrastructure products will deliver. However, to the best of our knowledge, no work in the literature – traditional software engineering and AOSE – explicitly deals with the identification of how the infrastructure impacts on the software engineering process. This appears very strange because the presence of a specific infrastructure could influence the engineering process. For example, an engineer could design or could not design specific functions according to the services / functionality provided by the infrastructure. This is specially true in the context of on-line engineering where the infrastructure's abstractions have a great impact on the engineering process.

In the last years, research on AO methodologies and multi-agent system (AO) infrastructures has developed along two opposite paths: while AO methodologies have essentially undergone a top-down evolution pushed by contributions from heterogeneous fields like human sciences, AO infrastructures have mostly followed a bottom-up path growing from existing and widespread (typically object-oriented) technologies. This dichotomy has produced a conceptual gap between the proposed AO methodologies and the agent infrastructures actually available, as well as a technical gap in the MAS engineering practice, where methodologies are often built *ad hoc* out of MAS infrastructures, languages and tools. This chapter presents the infrastructures research field in order to provide an overview of the state the art for both the traditional – often called middleware – and AO infrastructures. In addition, this chapter provides the meta-modelling representation of several AO infrastructures: by allowing structural representation of abstractions to be captured along with their mutual relations, meta-models make it possible to map design-time abstractions from AO methodologies upon run-time abstractions from MAS technologies, thus promoting a more coherent and effective practice in MAS engineering. On one hand, this approach allows software engineers to investigate the depth of the gap

between specific couples of methodology and infrastructure, so they could choose the one with the smaller gap. On the other hand, this approach allows software engineers to create new methods for filling the gap between methodologies and infrastructures.

So, the remainder of this chapter is structured as follows. Section 10.1 presents some general definition of infrastructures. Section 10.2 discusses the peculiarities of the infrastructures for MAS, while the relationships between infrastructures and Software Engineering and between infrastructures and AOSE are depicted respectively in Section 10.3 and Section 10.4. Then, Section 10.5 presents some example of agent-oriented infrastructures, finally conclusions are reported in Section 10.6.

10.1 Definitions

In its most general sense, 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 that an infrastructure is a persistent entity: once installed, an infrastructure typically survives the many systems it supports. Also, definitions (4) and (5) remark on 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 [70] 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. [205]:

“Agents in a MAS are expected to coordinate by exchanging services and information, to be able to follow complex negotiation protocols, to agree on commitments 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.

10.2 Infrastructures for MAS

With regard to MAS, infrastructure obviously plays a key role, given the potential complexity of both the system components (agents) and the component interplay (agent societies), and in this project will be defined as the set of services, conventions, and knowledge that support agents in coordinating by exchanging services and information, in following complex negotiation protocols, in agreeing on commitments and in performing other socially complex operations [148]. In a more abstract sense 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 some elements of “social control” – like coordination rules or security constraints – 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 [149].

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 could be good 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.

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 [48]. 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, 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.

10.2.1 Enabling vs. Governing Infrastructures

Infrastructures are useful to encapsulate and support critical features and properties of MAS; these properties typically concern the interaction dimension [148]. For this extent, current MAS infrastructures can be considered enabling infrastructures, 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 one of the most important infrastructure currently adopted for MAS development and deployment: JADE [7]. 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 MASs 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 is Electronic Institutions [139]: the social and normative capabilities required to infrastructures supporting eInstitutions go 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 of the specific agent model, the

TEAMCORE [207, 209] 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 [206]. Similar team-oriented capability has been added to RETSINA by enhancing its Individual Agent Architecture [74]: 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 suitable 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 in which to embody a conceptual framework that uniformly accounts for organisation, coordination and security of MAS altogether [150]. 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 the information system infrastructure that support the RBAC model [190], 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 [147, 214]: 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 coordination abstraction is promoted to a first class entity in coordinated systems, amenable to an explicit characterisation in terms of an interactive abstraction, with the goal of supporting the design of coordination models down to the deployment of coordination infrastructures. Coordination abstraction is the conceptual locus where coordination takes place. 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. An example of a powerful coordination abstraction is represented by *coordination artifact* [153]. Coordination artifacts are of particular interest in the context of agent societies, where they are usually exploited to achieve or maintain a global behaviour which is coherent with the society's social goal. As such, a coordination artifact is an essential abstraction for building

social activities, in that it is crucial both for enabling and mediating agent interaction, and for governing the social activities by ruling the space of agent interaction. Examples range from artifacts for concurrency management – such as semaphores, synchronisers, barriers, etc. – to artifacts for communication management – such as blackboards, event services – up to artifacts with articulated behaviours, such as workflow engines or auction engines. So, coordination artifacts generalise the common notion of coordination medium as coming from the field of coordination models and languages: simply put, coordination artifacts are the artifacts for MAS encapsulating the activity of MAS coordination. As such, they are the main tool for engineering the space of agent interaction, taking care of issues like concurrency, synchronisation, sharing of resources, and the like.

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: Computer Supported Cooperative Work (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 [193], 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 AO methodologies, since coordination abstractions – as they embody the social aspect of MAS – are meant to become explicitly subject to all the engineering stages, as it happens in SODA methodology (Chapter 14).

10.3 SE and Infrastructures

As mentioned above, an infrastructure – or *middleware*, as it is often called in traditional SE field – is a technical and social substrate that stabilises and rapidly enables instrumental (domain-centric, intentional) activity in a given domain [70]. Said another way, infrastructure solves typical, costly, commonly-accepted community problems in systematic and appropriate ways. So, infrastructure allows much greater community attention to unique, domain-specific activities.

For example, the construction of a distributed system is considerably more difficult than building a centralised or client/server system. This is because there are multiple points of failure in a distributed system, heterogeneous components need to communicate with each other through a network, which complicates communication and opens the door for security attacks. Infrastructure has been devised in order to conceal these difficulties from application engineers as much as possible [29]. Infrastructure resolves heterogeneity, and facilitates communication and coordination of distributed components.

Existing infrastructure products enable software engineers to build systems that are distributed across a local-area network, and they offer pre-built services that support, for example, off-the-shelf distributed transaction processing, security, and directory and nam-

ing services. The infrastructures also provide specialised components that let developers integrate many different legacy systems and design and deploy new business processes that integrate multiple distributed applications.

From the software engineering viewpoint the infrastructures present two key challenges illustrated in the following: the infrastructure selection (Subsection 10.3.1) and the impact of the infrastructures in the engineering process.

10.3.1 Infrastructure Selection

Two widely used but very different infrastructure styles (see Subsection 10.3.2) are object-oriented and message-oriented. Within each of these styles, there are multiple products to choose from. Moreover, any of these products may be used alone or in combination with other products. Thus the problem of infrastructure selection is increasingly important in the engineering of enterprise software systems [102].

Infrastructure selection is important for a number of reasons. It is an essential part of the way in which distributed systems get built, both by new development and by integration of existing applications and services. Moreover, infrastructure is a key enabling technology: it provides services, supports application functions and features, separates concerns, and integrates components [102]. In these roles, infrastructure interacts with and may impact on many other kinds of technologies, such as database systems, workflow engines, web servers, and applications. It further affects system architecture and development processes.

Infrastructure selection is also challenging, for these same reasons. Infrastructure selection can be an involved software (and systems) engineering process in its own right, with all the technological, organisational, economic, and political aspects that this may imply. Because of the central position and critical function of infrastructure, if it is selected or applied inappropriately, it can become a key disabling technology. Integration projects may present more or fewer challenges for infrastructure selection, depending on the diversity of systems and applications to be integrated. For new development, the constraints on infrastructure may be initially less restrictive, but infrastructure selection must be closely integrated with the design of other elements of the system, so selection can still be complex. In addition, the role of infrastructure selection must be defined with respect to the rest of the software life cycle. Infrastructure selection should generally be based on system requirements specifications (that account for both technological and organisational factors), but infrastructure selection should not be wholly determined by those specifications. Depending on circumstances, infrastructure selection may occur prior to, along with, or subsequent to system architectural specifications. Consequently, infrastructure selection may constrain, or be constrained by, architectural design.

Despite the potential variety of middleware selection processes, a number of guidelines or recommendations regarding infrastructure selection processes can be identified [102]:

- selection may be made based on a number of approaches to comparing and evalu-

ating infrastructure technologies;

- the frameworks, taxonomies, and criteria that are used for comparing and evaluating infrastructure must reflect enterprise-specific and project-relevant conditions;
- standards can be useful for enhancing communication among stakeholders as well as facilitating technical integration;
- the selection process should allow for the adoption of multiple alternative (or complementary) technologies in the solution to any particular problem.

Three other important factors should be considered in the infrastructure selection [156]:

- *Programming paradigm.* Adopting a new technology changes the concepts, the techniques and the programming styles used to create software and to reason about it (*paradigm shift*).
- *Development Process.* Adopting a new technology affects the actual software development activities and how they are related.
- *Economical Environment.* Adopting a new technology can benefit some stakeholders more than others, and a market situation can turn out to be more or less favorable to a new technology.

An example of an infrastructure selection process can be found in [113]. The authors present the i-Mate process that offers three tools to help organisations more effectively engineer Internet-based applications that require infrastructure technology. i-Mate provides a defined process for gathering, ranking, and weighting application requirements that the infrastructure must meet.

10.3.2 A Sketch of the State of the Art

Current middleware technologies fall into four broad categories: Corba-based technology, Java-based technology, messaging and integration technologies, proprietary technologies [113].

Corba-based technologies [141] enable transparent, synchronous inter-object communication across different processes, written in different languages, running on different hosts. These technologies hide the complexities of low-level networking details from the application programmer and provide easy integration across heterogeneous computing platforms. Further, IIOP (Internet Inter-ORB Protocol) enables integration of systems built on different Corba products.

Java-based technologies have evolved through Sun's Java 2 Enterprise Edition specification (J2EE) [204]. This defines the Enterprise Java Beans programming model, which is built on the Java Remote Method Invocation (RMI) communication protocol. It enables

communication between different Java components across different machines and simplifies many distributed-programming issues by incorporating transaction services, component life-cycle, integration and persistence services, and security.

Messaging and integration technologies include message-oriented middleware such as IBM's Websphere MQ family [94]. These widely deployed messaging products provide asynchronous communication models for autonomous systems to exchange messages in both point-to-point and multicast modes. Add-on services such as message transformation and translation engines simplify business integration by unifying message formats and offering event-triggering services.

Proprietary technologies include for example Microsoft .Net platform [122], which provides component-based technology for deploying applications on the Windows platform. .Net includes class libraries that give components access to base-level middleware services. All .Net compilers then generate a common intermediate code representation that is translated to machine code as the application executes. .Net also includes extensive support for Web services technologies, which provide a nonproprietary access mechanism for .Net applications.

10.4 AOSE & Infrastructures

The availability of 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 [148].

Infrastructure should provide flexible and robust abstractions to model and shape the agent interaction space, in accordance with the social and normative objectives of systems. As illustrated for instance by the **SODA** methodology (Chapter 13), coordination abstractions provided by a MAS infrastructure should represent the runtime embodiment of the same analysis / design abstractions used from the early stages of the MAS engineering process. Keeping abstractions alive along the whole engineering process is in fact essential to support advanced practices like *online engineering*: there is no viable way to evolve a system online when the design abstractions are no longer in place at runtime. Here, then, infrastructures should play a key role: they should provide MAS engineers with the same abstractions used in the analysis / design phases, as well as the tools to for their off-line / online development, deployment, monitoring and debugging.

Infrastructures also represent an effective approach to the general issue of formalisability of complex systems, which may come from either pragmatism or theoretical problems [153]. By their very nature, infrastructures intrinsically encapsulate key portions of systems—often in charge of the critical system behaviour. As a result, providing well-

specified infrastructures promotes the discovery and proof of critical system properties. Most notably, a system property can be assessed at design time through the formal definition of some design abstraction. Then, by ensuring compliance of the corresponding run-time abstraction provided by the infrastructure, such a property can be enforced at execution time and be automatically verified for any system based on the infrastructure.

Infrastructures, being the concrete realisation of a conceptual model with software artefacts, can help with technology transfer in all the three dimensions described above (the last three points in Subsection 10.3.1). In short [156],

- the paradigm shift or adaptation is eased by a well-defined set of APIs,
- infrastructure can effectively cooperate with tools to shape the development process,
- and a good quality, freely available infrastructure can rapidly increase the adoption of a new technology, thereby creating market opportunities for services and applications based on that technology.

While the above applies to infrastructure in general, agent-oriented infrastructure has some more peculiar features of its own. In the case of agent technology, the pivotal concept of *situated agent* brings about some interesting consequences when considered in the context of heterogeneous software integration. In complex applications, where a high degree of system integration is needed, a trend towards seamless situated agents can be observed; that is, a multi-agent system can be embedded in an application and work together with other sub-systems adopting different component models. In principle, this is in contrast with the *wrapper agent* approach, where every non-agent software entity or subsystem has to be hidden behind an agent.

From the perspective of a software agent, the outcome is that most interactions with non-agent software are modelled as interactions with the physical environment [156]. So, the balance between social and physical environment critically depends on the amount of agent wrapping performed on non-agent components. In a highly wrapped solution, such as the one specified in the FIPA Agent-Software Integration document [66], nearly all interactions are likely to occur through agent communication language (ACL).

Language is about representation, so while it is not possible to physically act on a language-level entity, often there is the need to talk about physical domain entities. This means that the actions and events that describe the physical situatedness of a software agent must also have representations at the language level. The most common way to achieve this is by combining ACL and domain ontologies—this is the approach followed by FIPA specifications, too. The greater balance between the social and the physical environment of an agent, spurred by the trend towards seamless situated agents, puts forth the role played by the dimension of *interaction*.

There, in fact, complexity is no longer simply bound to the granularity of the individual system's components, but rather to the many different ways in which components

relate to and interact with each others, and with the overall system, too. Managing and governing the dimension of interaction is exactly the goal of *coordination*, which has emerged as a fundamental research area in many several fields—from software engineering to artificial intelligence, from social theory to economics, from organisational theory to biology. Coordination is perceived as issue to be the main challenge to address and solve in order to actually obtain seamlessly situated MASs; the complex spectrum that spans from purely linguistic interaction to lower level physical-like perception must be covered and effectively exploited to really connect agents with their social and physical environment.

10.4.1 Coordination, Organisation and Security

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 [149]. 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 cornerstones 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 [148].

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 [144]. 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 [149].

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 [145], 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 [150]. 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 in social activities; this participation includes accessing and using the coordination artifacts as part of organisation resources.

10.5 AO Infrastructures

This section presents some agent-oriented infrastructures: **JADE** (Subsection 10.5.1), **TuCSon** (Subsection 10.5.2), **CARTAgO** (Subsection 10.5.3) and **TOTA** (Subsection 10.5.4). Each infrastructure is presented by means of its meta-model composed by the static and dynamic models as discussed in Chapter Section 5.3.

10.5.1 JADE

JADE (Java Agent DEvelopment Framework) [6, 7] is a software framework fully implemented in the Java language, and provides an agent-oriented infrastructure. It is based on a middleware compliant with the FIPA specifications and offers a set of graphical tools that supports the debugging and deployment phases.

Static Model

Agents are the main entity in **JADE**, since it is an agent-oriented framework to develop MASs. In Figure 10.1 is depicted a fragment of the general version of the meta-model of JADE, extracted from [172]. The class *AgentJ* represents the agent: each agent has an associated *Agent State*, a *Scheduler*, and a *Message Queue*. An agent can be in different states, that represent the possible states listed by FIPA specifications: ACTIVE, DELETED, IDLE, INITIATED, SUSPENDED, TRANSIT, WAITING. Agents can move from one state to another according to the admitted transitions: the possible behaviours can be represented by a finite state machine. The scheduler is implemented by the base Agent class, it is hidden to the programmer, and carries out a round-robin non-preemptive scheduling policy among all behaviours available in the ready queue. Every behaviour is executed until the behaviour releases the control. The message queue is a sort of mailbox associated to every agent: it stores all the messages (written using the Agent Communication Language) sent by other agents and that have to be read by the agent. In the class diagram representing the meta-model, the class Message Queue is composed by a set of *ACL Messages*.

Behaviour is used to model a generic task, and is specialized into *CompositeBehaviour* and *SimpleBehaviour*. *SimpleBehaviour* models simple tasks, namely tasks that are not decomposed into sub-tasks. It is in turn specialized into *OneShotBehaviour* and *CyclicBehaviour*, respectively used to represent tasks to be executed only once and cyclic tasks that are restarted after finishing their execution cycle. *CompositeBehaviour* models complex tasks, that are made up by composing a number of other tasks: in order to represent the necessity of composition there is an aggregation between *CompositeBehaviour* and *Behaviour*, with cardinality 0..*. There are three specializations of the composite behaviours: *FSMBehaviour*, *SequentialBehaviour*, and *ParallelBehaviour*. *FSMBehaviour* is used when the complex task is composed by tasks corresponding to the states of a Finite State Machine; *SequentialBehaviour* is a classical sequential composition of sub-tasks; *ParallelBehaviour* allows the definition of concurrency, where tasks are executed in virtual parallelism.

Figure 10.2 shows a detail of the JADE meta-model focused on the Agent, the key concept of the architecture. Every agent has one or more associated AID (Agent Identifier) that can be used to reference an agent, and whose role is describing agents. There is also an association towards the class Codec, which represents the tool used to codify and decodify ACL messages and is internally called by the JADE framework. A set of agents

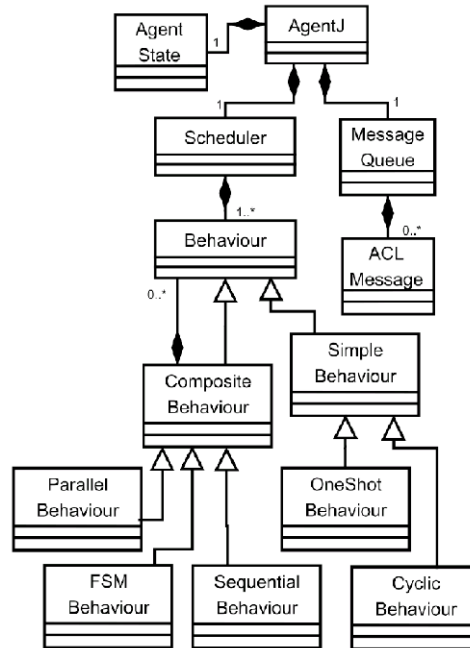


Figure 10.1: JADE Meta-model

share an *Agent-Toolkit*, namely the place where they live and execute: a typical example of agent toolkit is the JADE container. Agents also have an associated Ontology, needed to give semantics to the content of the ACL messages agents exchange.

JADE is provided with a content reference model (Figure 10.3), which is needed to perform the proper semantic checks on a given content expression, by identifying all possible elements in the domain of discourse.

The classification is derived from the ACL language specifications. A *Predicate* is an expression that says something about the status of the world and can be either true or false. A *Term* is an expression identifying both abstract and concrete entities, that exist in the world and that the agents talk and reason about. The class *Term* is abstract, and is refined into *IRE*, *Concept*, *Primitive*, *Aggregate*, and *Variable*. An *IRE* (Identifying Referential Expression) is an expression that identifies the entities for which a given predicate is true. A *Concept* is an expression that indicates entities with a complex structure that can be defined in terms of slots. A *Primitive* represents atomic entities such as strings and integers. An *Aggregate* is an expression indicating entities that are a group of other entities. A *Variable* is an expression used to indicate a generic element not known a-priori. As well as the class *Term* is used to represent entities, the class *ContentElement* is introduced to express the content of an ACL message, and the class *Predicate* derives from it. *ContentElementList* is used to indicate a list of content elements. The class *AgentAction* derives from both *ContentElement* and *Concept*. JADE ontologies have to deal with predicates, concepts, and agent actions.

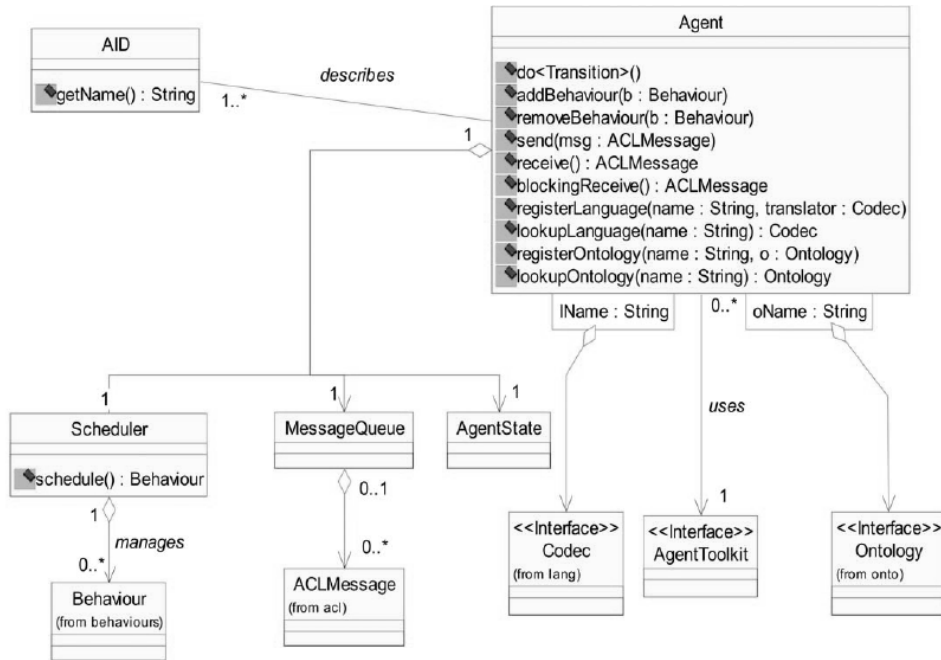


Figure 10.2: JADE Meta-model: focus on agent [119]

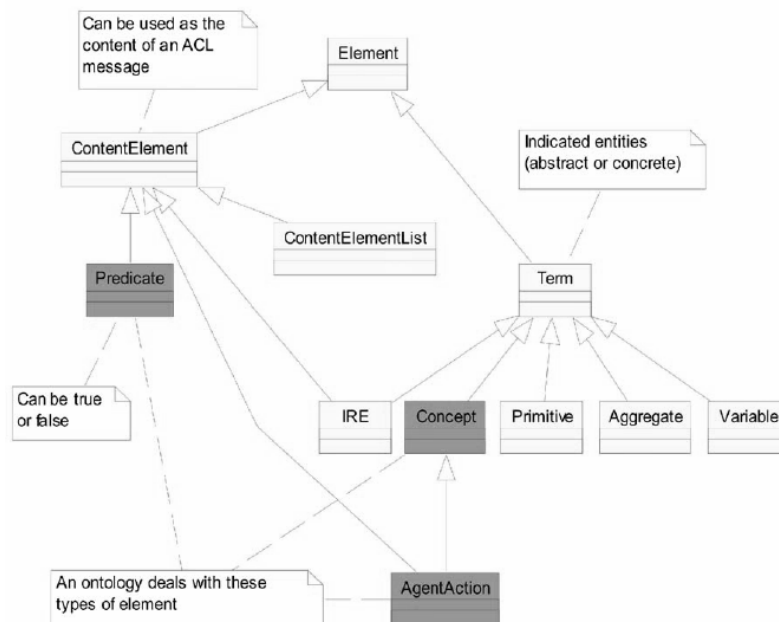


Figure 10.3: JADE Meta-model: focus on content

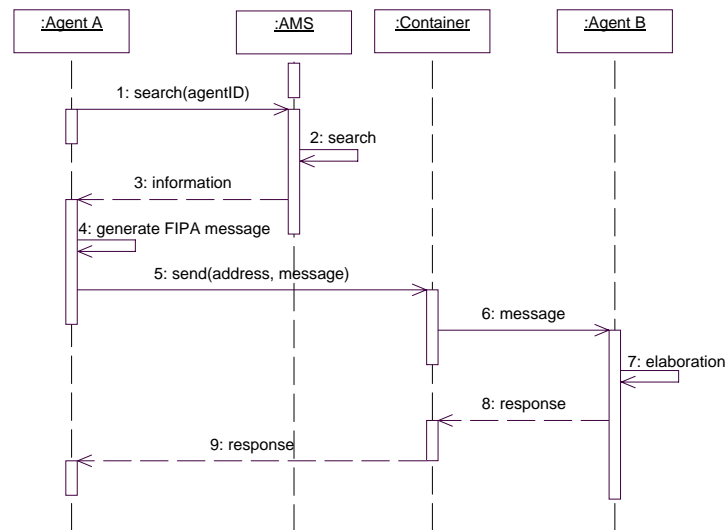


Figure 10.4: Message passing

Dynamic Model

From the functional point of view, **JADE** provides the basic services necessary for distributed peer-to-peer applications in a fixed and mobile environment. **JADE** allows each agent to dynamically discover other agents and to communicate with them according to the peer-to-peer paradigm. From the application point of view, each agent is identified by a unique name and provides a set of services: it can register and modify its services and/or search for agents providing given services, it can control its life cycle and, in particular, communicate with all other peers.

The first service / agent is the *Agent Management System* (AMS) whose responsibility is to manage the life cycle of all agents in the platform, and to provide a white pages service associating a logical agent identifier to status and address information [183] as illustrated in Figure 10.4. The agent A asks the AMS for address of a particular agent B. The AMS executes an internal search and produces a result for A. Now A is able to send a FIPA message to B through the infrastructure layer.

In Figure 10.5 is an example of the interaction among an agent and AMS: the AMS must authorise each operation that the agent makes inside the infrastructure. In the specific case of the Figure 10.5 an agent A asks the AMS for suspension of another agent B. If the AMS authorises this suspension the message is propagated through the infrastructure layer (indicated as *Container* in the figure) to agent B (part a) of the figure); otherwise A is informed that the suspension of B is not possible.

The second service / agent is the *Directory Facilitator* (DF), that works as a yellow pages service with basic matchmaking capabilities. A special instance of this agent – named the *Default DF* – must be available in a FIPA compliant agent platform. An

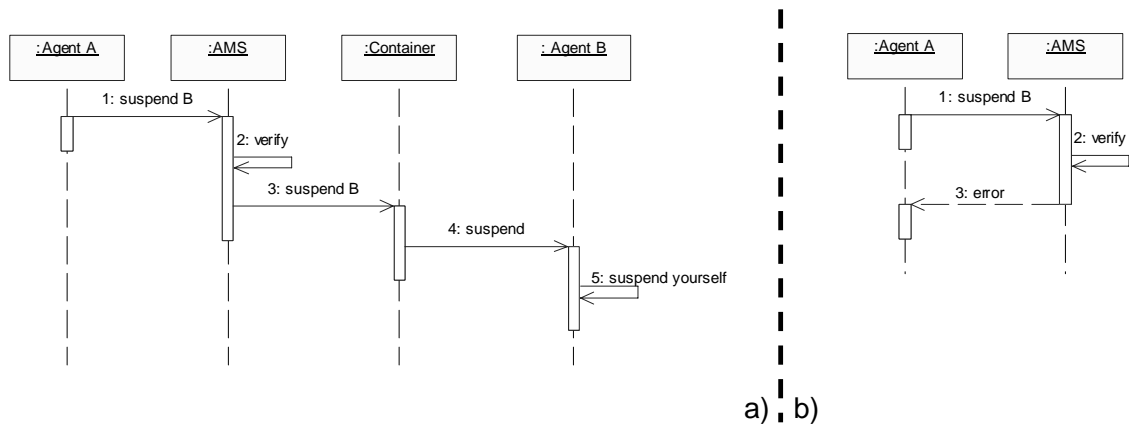


Figure 10.5: The request of a suspension of an agent: a) the authorised case, b) the non-authorized case

example of this is reported in Figure 10.6 where the agent A contacts the Default DF searching an agent that provides a specific service. The Default DF executes an internal search and produces the information for A. So A is now able to send its request to the agent B.

10.5.2 TuCSon

TuCSon (Tuple Centres Spread over Networks) [157, 212] is an infrastructure providing services for the communication and coordination of distributed / concurrent independent agents.

Static Model

TuCSon's meta-model is depicted in Figure 10.7 [124].

In detail, **TuCSon** supports agent communication and coordination via *tuple centres* coordination media [71]: these are shared & reactive information spaces, distributed over the infrastructure *nodes*. In turn, this inducts a *topology* over the *network*. Agents access tuple centres associatively, by writing (*out*), reading (*rd*, *rdp*), and consuming (*in*, *inp*) *tuples* – i.e., ordered collections of heterogeneous information chunks – via the above coordination primitives.

A tuple centre is a tuple space enhanced with the notion of behaviour specification. More precisely, a tuple centre is a coordination abstraction perceived by the interacting entities as a standard tuple space [72], but whose behaviour in response to events can be defined so as to embed the coordination laws. So, defining a new behaviour for a tuple centre basically amounts to specifying state transitions in terms of *reactions* to *events* [157]. In particular, reactions are specified in **TuCSon** via the **ReSpecT** (Reaction

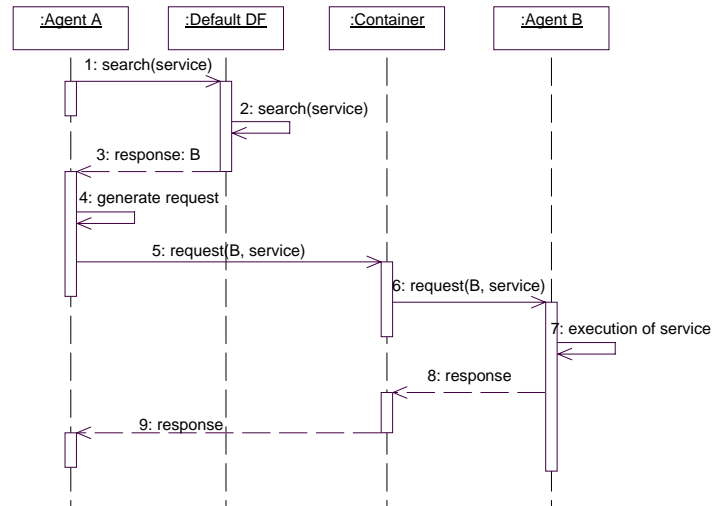


Figure 10.6: Discovery of a service

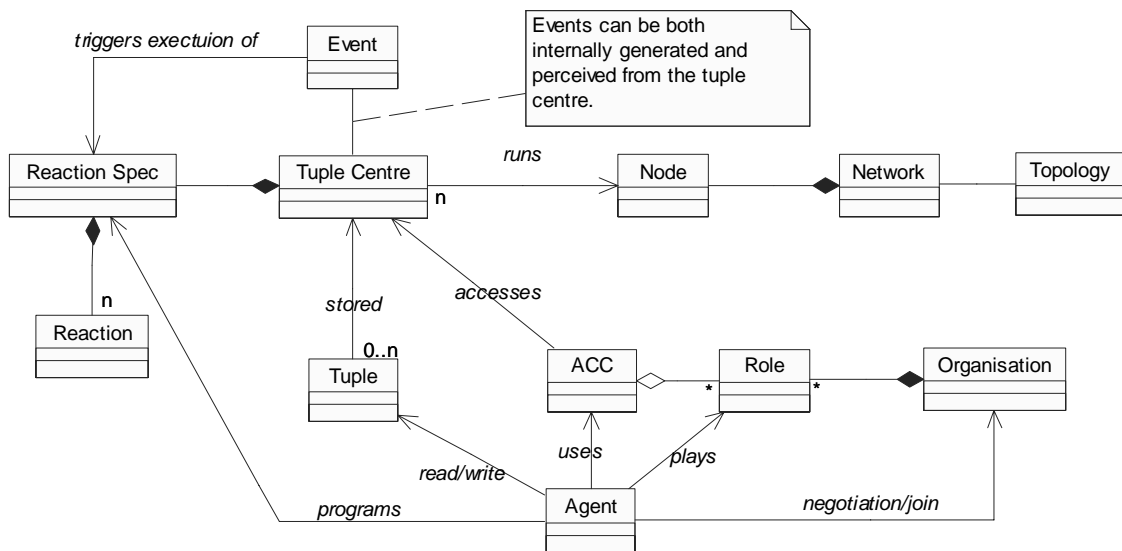


Figure 10.7: TuCSoN Meta-model

Specification Language) language [146]: a reaction is defined as a set of non-blocking operations [157], and has a success/failure transactional semantics: a successful reaction may atomically produce effects on the tuple centre state, a failed reaction yields no result at all. Typically, a tuple centre contains a set of reactions (*reaction spec* in Figure 10.7), each tied to a specific event: the same event could trigger multiple, different reactions. Tuple centres are connected to each other through *link* operations, having the same form and a similar semantics as **TuCSon** coordination primitives, but invoked by successful reactions rather than by agents [146].

The Agent Coordination Context (*ACC*), introduced in [145] as the conceptual place where to set the boundary between the agent and the environment, encapsulates the interface enabling agent actions and perceptions inside the environment. More precisely, an ACC (*i*) works as a model for the agent environment, by describing the environment where an agent can interact, and (*ii*) enables and rules the interactions between the agent and the environment, by defining the space of the admissible agent interactions. The ACC dynamics is characterised by two basic steps: *negotiation* and *use*. In fact, an ACC is meant to be first negotiated by the agents with the MAS infrastructure, in order to start a working session inside an *organisation*. To this end, the agent specifies which *roles* to activate: if the agent request is compatible with the (current) organisation rules, a new ACC is created, configured according to the characteristics of the specified roles, and is released to the agent for active playing inside the organisation. The agent can then use the ACC to interact with other agents in the organisation, and with the organisation environment, by performing the actions and activating the perceptions made possible by the ACC.

Dynamic Model

The ACC dynamics is characterised by two basic steps: *negotiation* and *use*. In fact, an ACC is meant to be first negotiated by the agents with the MAS infrastructure in order to start a working session inside an *organisation* (Figure 10.8 part a)). To this end, the agent specifies which *roles* to activate: if the agent request is compatible with the (current) organisation rules, then a new ACC is created and configured according to the characteristics of the specified roles, and is released to the agent for active playing inside the organisation (Figure 10.8 part a)). However, an ACC could be re-negotiated by an agent during its life cycle (Figure 10.8 part a)). In this case the agent contacts the organisation asking for the activation of new different roles. The organisation evaluates the request and if this last is compatible with the organisation rules then the ACC is updated and released again to the agent.

The agent can then use the ACC to interact with other agents in the organisation, and with the organisation environment, by performing the actions and activating the perceptions made possible by the ACC. In particular each action an agent performs must be authorised by ACC (Figures 10.10 and 10.9): if the action is not allowed according

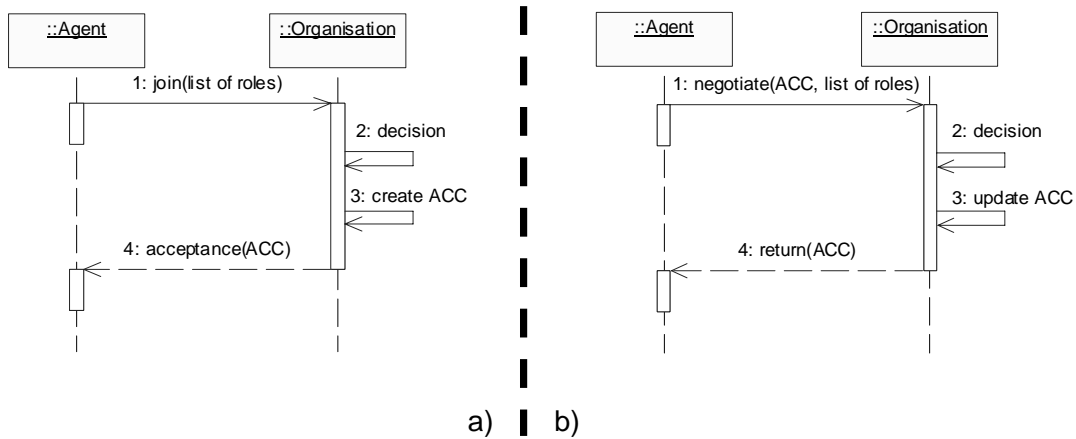


Figure 10.8: Agent joins organisation (a) and agent re-negotiate ACC (b)

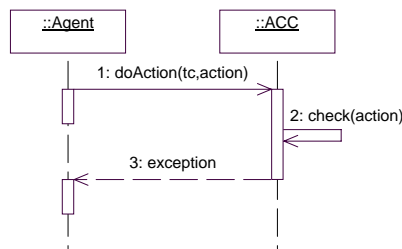


Figure 10.9: Execution of action not authorised by ACC

to the ACC policy, an exception is generated, preventing action execution (Figure 10.9); otherwise, the action is executed.

Typically an agent action is the execution request of a coordination operation. The coordination operations are the tuple centre coordination primitives, for inserting, reading and retrieving tuples from the tuple centre (Figure 10.10). The execution of the incoming action triggers one or more reactions that are executed inside the tuple centre. The execution of reactions generates a partial response in the tuple centre, and this could trigger other reactions that are executed and generate the final result inside the tuple centre (Figure 10.10 points 5-8). The result is sent to the ACC, and the agent retrieves it from the ACC.

A particular case of the tuple centre coordination primitives are the primitives for inspecting / changing the behaviour specification of a tuple centre (Figure 10.11). When a primitive of this kind is executed in the tuple centre no reactions are triggered and the tuple centre sends immediately the generated result to the ACC.

A tuple centre is also able to perceive events generated in the network (for example events generated by other tuple centres or agents). When an event is perceived by a

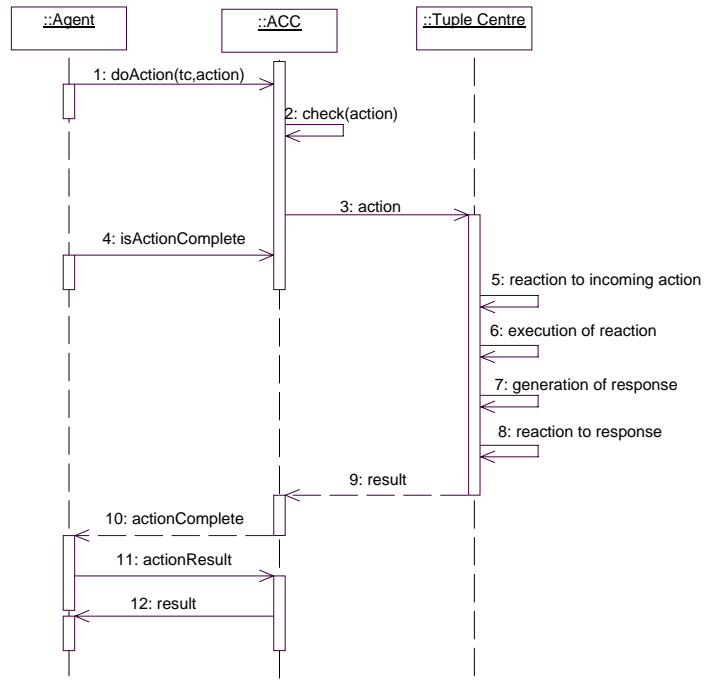


Figure 10.10: Execution of action authorised by ACC

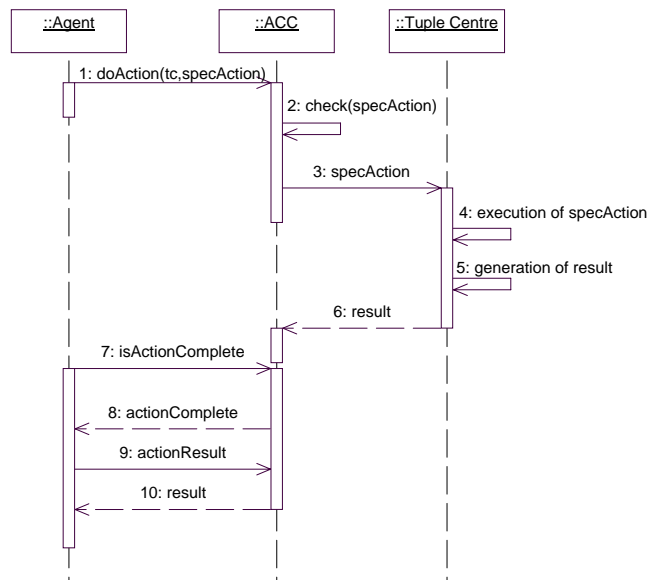


Figure 10.11: Programming of a Tuple Centre

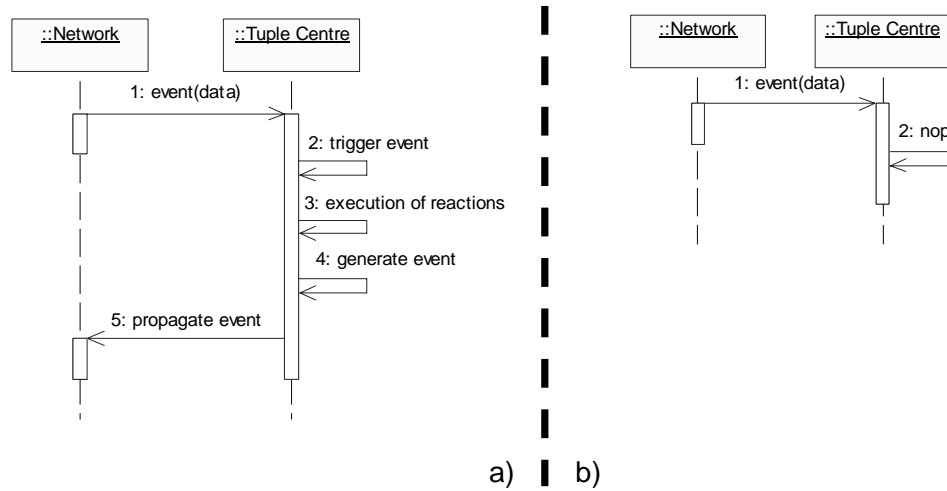


Figure 10.12: Managing of event

tuple centre, it could react by executing the triggered reactions (Figure 10.12 part a), and eventually it could generate and propagate an event in the network; otherwise no reaction could be triggered by the event (Figure 10.12 part b).

10.5.3 CArtAgO

CArtAgO [23, 181] is a framework for developing artifact-based working environments for multi-agent systems. The framework is based on the notion of artifact, as a basic abstraction to model and engineer objects, resources and tools designed to be used and manipulated by agents at runtime to support their working activities, in particular the cooperative ones. **CArtAgO** makes it possible for MAS engineers to design and develop suitable kinds of artifacts, and to extend existing agent platforms with the possibility to create artifact-based working environments, and programming agents to exploit them.

Static Model

The abstract architecture of **CArtAgO** (Common Artifact for Agents Open environment) [181, 23] is composed of three main elements (see Figure 10.13 [124]): *(i) agent bodies* – as the entities that make it possible to situate agents inside the working environment; *(ii) artifacts* – as the basic building blocks to structure the working environment; and *(iii) workspaces* – as the logical containers of artifacts, aimed at defining the topology of the working environment.

Agent bodies. The agent body contains *effectors* to perform *actions* upon the working environment, and a dynamic set of *sensors* to collect *events* from the working environment. Agents interact with their working environment by “piloting” their bodies: they

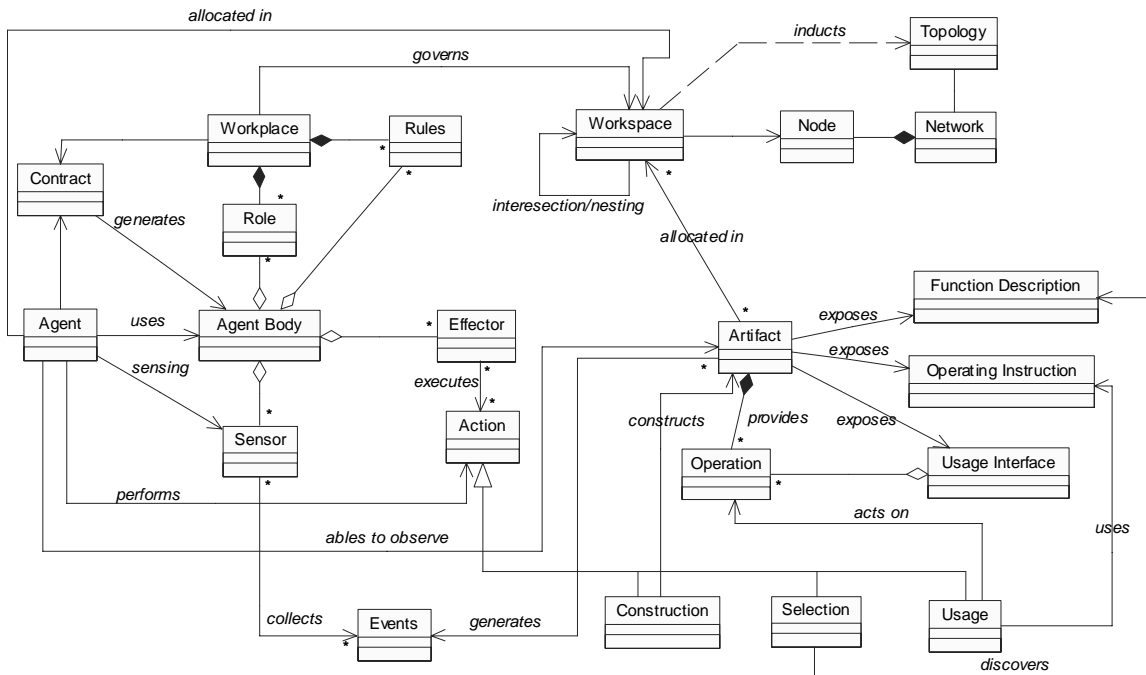


Figure 10.13: CArtAgO Meta-model

execute actions to *construct*, *select* and *use* artifacts, and perceive the *observable events* generated by artifacts.

Artifacts. Artifacts are the basic bricks managed by CArtAgO: agents use artifacts by triggering the execution of *operations* listed in the artifact *usage interface*. The execution of an operation typically causes the update of the internal state of an artifact, and the generation of one or more observable events: these are then collected by the agent sensors and perceived by means of explicit sensing actions. In order to support a rational exploitation of artifacts by intelligent agents, each artifact is equipped with a *function description*, i.e. an explicit description of the functionalities it provides, and *operating instructions*, i.e. an explicit description of how to use the artifact to get its function.

Workspaces. Artifacts are logically located in workspaces, which define the *topology* of the working environment.

In addition, CArtAgO also introduces the concept of *workplace* as an organisational layer on top of workspaces. More precisely, a workplace is the set of *roles* and *organisational rules* being in force in a workspace: there, *contracts* define the norms and policies that rule agent access to artifacts and allow the generation of agent bodies. So, for instance, an agent may or may not be granted permission to use some artifacts or to execute some specific operations on selected artifacts depending on the role(s) that the agent is playing inside the workplace [179].

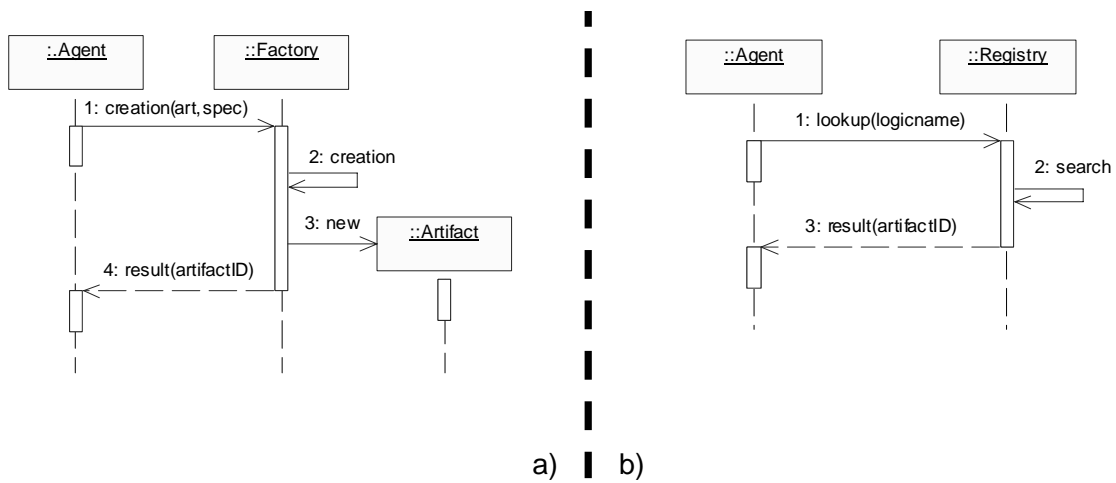


Figure 10.14: Agent creates a new artifact (a) and agent searches an artifact

Dynamic Model

CARTAgO provides two *default artifacts* in each workspace in order to support agents in the creation and discovery of artifacts. In particular the *Factory Artifact* creates new artifacts and the *Registry Artifact* provides a white page services.

In order to create an artifact, an agent uses the Factory Artifact (Figure 10.14 part a) providing to it information about the new artifact: a logic name, the template that identifies the type of the artifact to be created, the initial configuration parameters and optionally the workspace where the artifact should be created. The action can fail if the template is unknown or the artifact instantiation is not completed due to some kind of problem. In order to search an artifact, an agent uses the Registry Artifact (Figure 10.14 part b) providing the logic name of the artifact. Then the Registry Artifact sends the artifact identifier to the agent.

In order to execute an operation over an artifact, an agent must first join the workspace where the artifact is allocated, then a workspace produces the relative agent body (Figure 10.16 points 1-3). By means of agent body the agent is able to use an artifact specifying the artifact identifier, and the operation name (Figure 10.16 points 5-7).

The artifact executes the requested operation, eventually changes its state and the generates one or more events propagated to the agent body (Figure 10.16 points 5-11). The action can fail either because the specified artifact is not available, or because the operation cannot be executed since it is not part of artifact usage interface (as illustrated in Figure 10.15). Action success means that the execution of the specified operation has been successfully triggered. After that the agent is able to access the operation results by means of a sensing operation (Figure 10.16 points 13-14).

However an agent could observe all the events produced by a specific artifact without using it. In this case the agent manifests this intention by means of the *focus* action: each

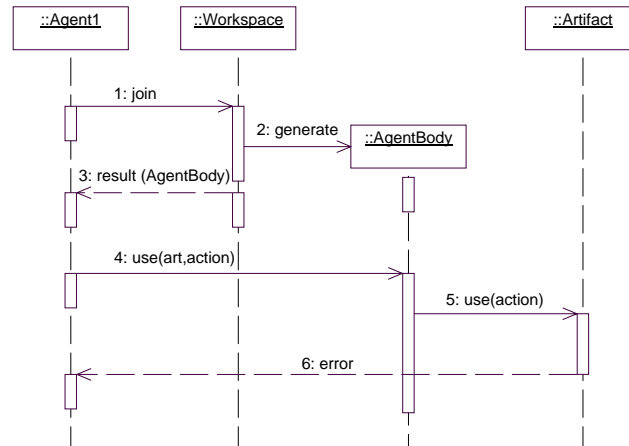


Figure 10.15: Problems in the use of artifact

time the artifact generates an event, this event is propagated to the agent (Figure 10.16 points 4,12).

10.5.4 TOTA

TOTA (Tuples On The Air) [114, 115] is an infrastructure for multi-agent coordination, in distributed computing scenarios.

Static Model

A meta-model of the infrastructure is presented in Figure 10.17 [124]. **TOTA** assumes the presence of a network of possibly mobile *nodes*, each running a *tuple space* [71]: each agent is supported by a local middleware and has only a local (one-hop) perception of its environment. Nodes are connected only by short-range *network* links, each holding references to a (limited) set of *neighbour* nodes: so, the *topology* of the network, as determined by the neighbourhood relations, may be highly dynamic.

In **TOTA**, tuples are not associated to a specific node (or to a specific data space) of the network: rather, they are “injected” in the network by an agent from some node, then autonomously propagate hop-by-hop, diffuse, and evolve according to specified propagation patterns. Thus, **TOTA** tuples form a sort of spatially-distributed data structure, that can be used to acquire contextual information about the environment and to support the mechanisms required for stigmergic interaction [167]. More precisely, **TOTA** distributed tuples $T=(C,P,M)$ are characterised by a *content* C , a *propagation rule* P , and a *maintenance rule* M : the content C is an ordered set of typed fields representing the information carried by the tuple, the propagation rule P determines how the tuple propagates across the network (called “migration” in the Figure 10.17) and how the tuple content should

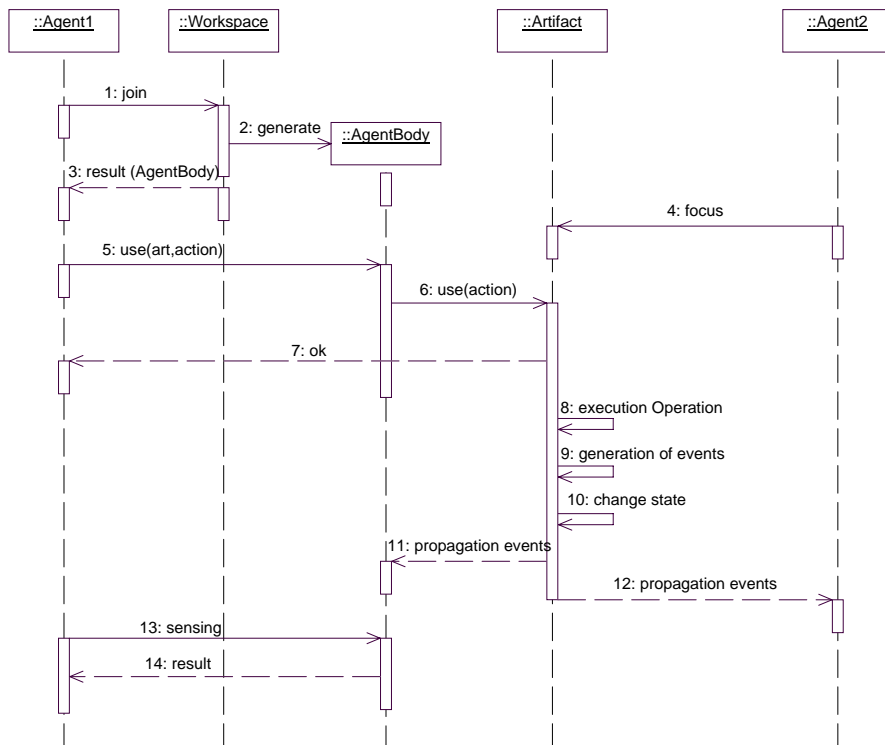


Figure 10.16: Agent uses an artifact

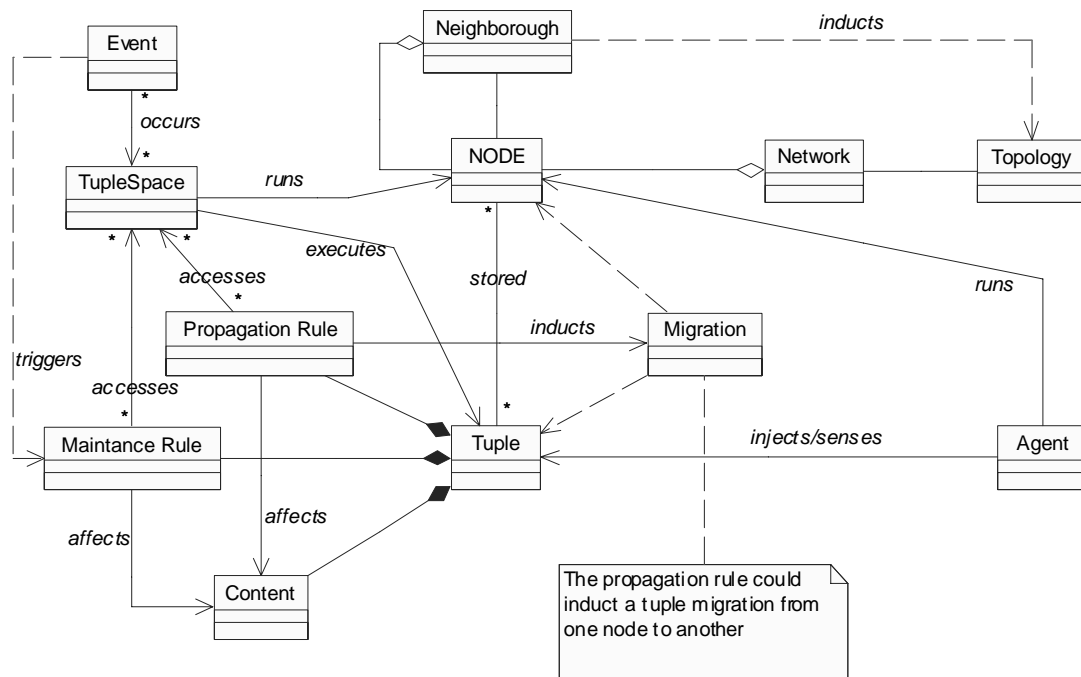


Figure 10.17: TOTA Meta-model

change while the tuple is propagated; finally, the maintenance rule *M* determines how a tuple distributed structure should react to *events* occurring in the environment. Specifying the tuple propagation rule includes determining the “scope” of the tuple and how such propagation is affected by the presence or absence of other tuples in the system. In turn, events handled by the maintenance rule can range from simple time alarms, to changes in the network structure: the latter kind of event is of fundamental importance to preserve a coherent structure of the environment properties represented by tuple fields.

Dynamic Model

In **TOTA** tuples are injected in the network and can autonomously propagate, diffuse, and evolve in the network according to specified patterns. Figure 10.18 shows an example of the tuple injection: an agent injects a tuple in the local tuple space when executes the propagation stored inside the tuple and propagates the tuple according to the propagation rule. The execution of the propagation rule could affect the content of the tuple (useful in the context of stigmergic coordination).

The *read* action accesses the local **TOTA** tuple space and returns a collection of the tuples locally present in the tuple space and matching the template tuple passed as parameter (Figure 10.19 points 1-3). The *readOneHop* action returns a collection of the tuples present in the tuple spaces of the node’s one-hop neighborhood and matching the

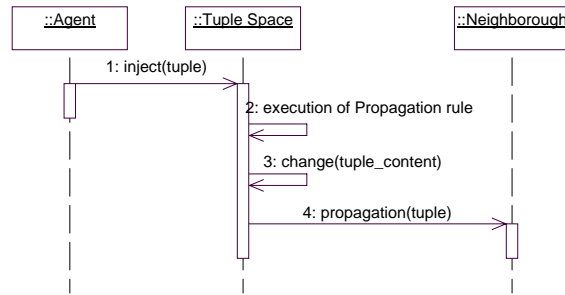


Figure 10.18: Agent injects a tuple

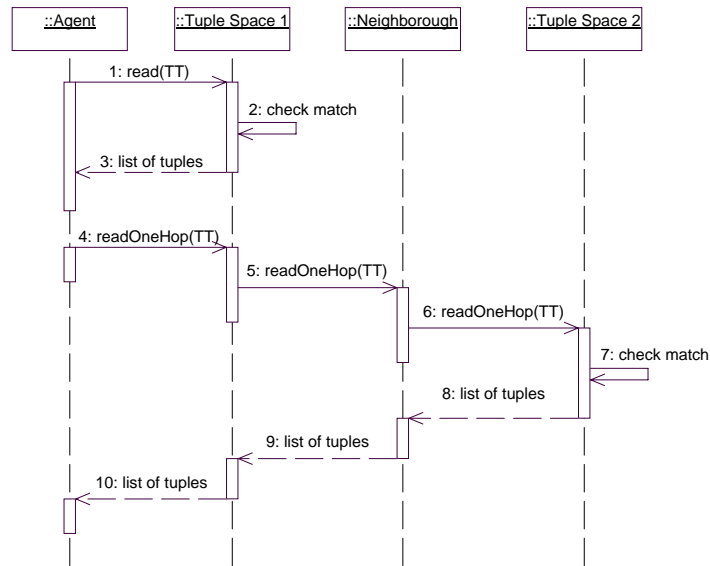


Figure 10.19: Agent reads tuples

template tuple (Figure 10.19 points 4-10).

As mentioned above, the maintenance rule M determines how a tuple’s distributed structure should react to events occurring in the environment. These types of event can be simple time alarms or they can be events associated to changes in the network structure. To this end, **TOTA** supports tuples’ propagation actively and adaptively: by constantly monitoring local events, the network local topology and the income of new tuples, the infrastructure automatically re-shapes tuples and their distributed structure whenever appropriate with respect to the maintenance rule (Figure 10.20). In particular the tuple space first executes all the maintenance rules associated to the stored tuples, changes the tuples’ contents according to maintenance rules and if necessary it propagates the tuples to the other tuple spaces.

In addition, *subscribe* and *unsubscribe* operations are defined in order for agent to

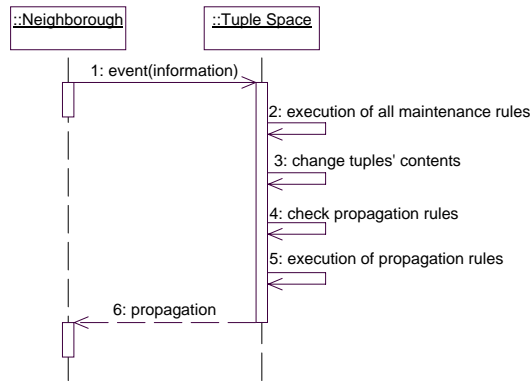


Figure 10.20: Tuple space reacts to events

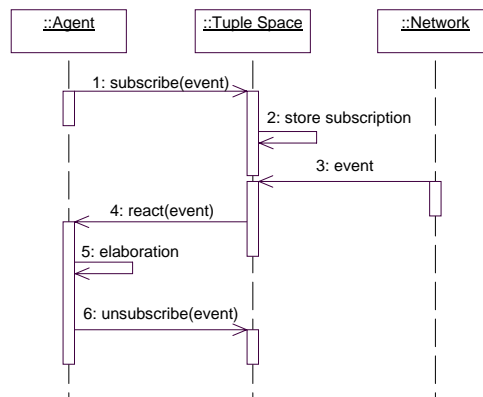


Figure 10.21: Agent subscribes and unsubscribes event

handle events (Figure 10.21). These operations rely on the fact that any event occurring in **TOTA** (including: arrival of new tuples, connection and disconnection of peers, system-level events) can be represented as a tuple. Thus: the subscribe operation associates the execution of a reaction method in the agent in response to the occurrence of events matching the template tuple passed. Specifically, when a matching event happens, the tuple space sends the matching event to the agent. The unsubscribe operation removes matching subscriptions.

10.6 Infrastructures: Summing up

MAS infrastructures play a fundamental role in the engineering of complex software systems. 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.

The importance of infrastructure is even more evident when considering agents and MAS as the main paradigm, whose typical scenarios involve complex systems composed by a multitude of distributed active entities organised according to some model, immersed in specific software / hardware environments, executing some kind of individual and collective tasks that require their fruitful interaction and coordination. Complexity of such scenarios can be framed at different levels of abstraction: from concurrency and distribution (control, space, time, . . .), to system / environment openness in terms of dynamism, heterogeneity, unpredictability, and so on. As in the case of human societies, infrastructures in MAS are meant to play a fundamental role in governing such complexity, factorising critical issues of systems in terms of services at different levels, from sustainability to security, from communication to coordination and organisation, and so on.

Currently, most of the available state of the art MAS infrastructures have been conceived and designed in academic contexts, and can be still considered in their infancy, in particular with respect to mainstream infrastructures. Among the others, such an immaturity affects three different aspects: *(i)* abstractions and services provided *(ii)* available tools and technologies *(iii)* integration with agent-oriented methodologies.

Then, the research on MAS infrastructure models and supporting tools, on the integration with agent-based methodologies, and on architectures and technologies used for MAS development is considered essential for advancing the state of the art on software engineering, in particular for what concerns complex software systems.

Part IV

Representation Complexity

11

Complex Systems

This chapter presents complex systems. Roughly speaking, a system is a collection of interacting elements making up a whole such as, for instance, a mechanical clock. While many systems may be quite complicated, they are not necessarily considered to be complex [11]. There is no precise definition of complex system: a complex system is any system featuring a large number of interacting components, whose aggregate activity is nonlinear and typically exhibits hierarchical self-organisation under selective pressures. In order to give a physical context to the definition, we should qualitatively discuss some typical systems that may be denoted truly complex.

The various branches of science offer us a large number of examples, some of which turn out to be rather simple, whereas others may be called truly complex. Let us start with a simple physical example. Granular matter is composed of many similar granules. Shape, position, and orientation of the components determine the stability of granular systems. The complete set of the particle coordinates and of all shape parameters defines the actual structure. Furthermore, under the influence of external force fields, the granules move around in quite an irregular fashion, whereby they perform numerous more or less elastic collisions with each other. A driven granular system is a standard example of a complex system. The permanent change of the structure due to the influence of external fields and the interaction between the components is a characteristic feature of complex systems. Another standard complex system is Earth's climate, encompassing all components of the atmosphere, biosphere, cryosphere, and oceans and considering the effects of extraterrestrial processes such as solar radiation and tides.

In biology, we are again dealing with complex systems. Each animal consists of various strongly interacting organs with an enormous number of complex functions. Each organ contains many partially very strong specialised cells that cooperate in a well-regulated fashion. Probably the most complex organ is the human brain, composed of millions nerve cells. Their collective interaction allows us to recognise visual and acoustic patterns, to speak, or to perform other mental functions. Each living cell is composed of a complicated nucleus, ribosomes, mitochondria, membranes, and other constituents, each of which contain many further components. At the lowest level, we observe many simultaneously acting biochemical processes, such as the duplication of DNA sequences or the

formation of proteins. This hierarchy can also be continued in the opposite direction. Animals themselves form different kinds of societies. Probably the most complex system in our world is the global human society with its numerous participants, its capital goods (such as machines, factories, and research centers), its natural resources, its financial and political systems, which provides us with another large class of complex systems.

In the software world, complex systems consist again of a number of related subsystems organised in a hierarchical fashion [99]. At any given level, subsystems work together to achieve the functionality of their parent system. Moreover, within a subsystem, the constituent components work together to deliver the overall functionality. Thus, the same basic model of interacting components, working together to achieve particular objectives occurs throughout the system: each component can be thought of as achieving one or more objectives. A second important observation is that complex systems have multiple loci of control: “real systems have no top”. Applying this philosophy to objective-achieving decompositions means the individual components should localise and encapsulate their own control. For the active and autonomous components to fulfill both their individual and collective objectives, they need to interact. However the system’s inherent complexity means it is impossible to a priori know about all potential links: interactions will occur at unpredictable times, for unpredictable reasons, between unpredictable components, i.e., the interactions *emerge*. For this reason, it is futile to try and predict or analyse all the possibilities at design time. It is more realistic to endow the components with the ability to make decisions about the nature and scope of their interactions at runtime.

This discussion has highlighted two important characteristics of the complex systems: the complex system can typically be represented as hierarchies and the complex systems have typically an emergent behaviour. So, the remainder of this chapter is organised as follows: Section 11.1 introduces the complex software systems and some of their features, Section 11.2 presents the hierarchical software system, and Section 11.3 presents the self-organising systems. Section 11.4 propose an example of hierarchical self-organisation system: the holonic systems. Finally Section 11.5 reports a summary of this chapter.

11.1 Software Systems and Complexity

Software entities are more complex for their size than perhaps any other human construct because no two parts are alike [100]. In this respect, software systems differ profoundly from computers, buildings, or automobiles, where repeated elements abound. Digital computers are themselves more complex than most things people build: they have very large numbers of states. This makes conceiving, describing, and testing them hard.

Software systems have orders-of-magnitude more states than computers do [100]. Likewise, a scaling-up of a software entity is not merely a repetition of the same elements in larger sizes; it is necessarily an increase in the number of different elements. In most cases, the elements interact with each other in some nonlinear fashion, and the complexity of

the whole increases much more than linearly. The complexity of software is an essential property – i.e., the difficulties are inherent in the nature of the software – not an accidental one—difficulties that attend the software production but are not inherent. Hence, descriptions of a software entity that abstract away its complexity often abstract away its essence [101].

For three centuries, mathematics and the physical sciences made great strides by constructing simplified models of complex phenomena, deriving properties from the models, and verifying those properties by experiment. This paradigm worked because the complexities ignored in the models were not the essential properties of the phenomena. It does not work when the complexities are the essence.

Many of the classic problems of developing software products derive from this essential complexity and its nonlinear increases with size. From the complexity comes the difficulty of communication among team members, which leads to product flaws, cost overruns, and schedule delays. From the complexity comes the difficulty of enumerating, much less understanding, all the possible states of the software, and from that comes the unreliability. From the complexity of structure comes the difficulty of extending programs to new functions without creating side effects [100]. From the complexity of structure come the security trapdoors. Not only technical problems, but management problems as well come from the complexity. It makes overview hard, thus impeding conceptual integrity. It makes it hard to find and control all the software lifecycle.

The next subsection presents some features of complex software systems.

11.1.1 Features of Complex Software Systems

Industrial-strength software is complex: it has a large number of parts that have many interactions [99]. Moreover this complexity is not accidental, it is an innate property of large systems. Given this situation, the role of software engineering is to provide structures and techniques that make it easier to handle complexity. Fortunately for designers, this complexity exhibits a number of important regularities [99]:

- Complexity frequently takes the form of a *hierarchy*. That is, a system composed of interrelated subsystems, each of which is in turn hierarchic in structure, until the lowest level of elementary subsystem is reached. The precise nature of these organizational relationships varies between subsystems; however, some generic forms (such as client/server, peer, team, and so forth) can be identified. These relationships are not static: they often vary over time.
- The choice of which components in the system are primitive is relatively arbitrary and is defined by the observer's aims and objectives.
- Hierarchic systems evolve more quickly than non hierarchic ones of comparable size (that is, complex systems will evolve from simple systems more rapidly if there are clearly identifiable stable intermediate forms than if there are not).

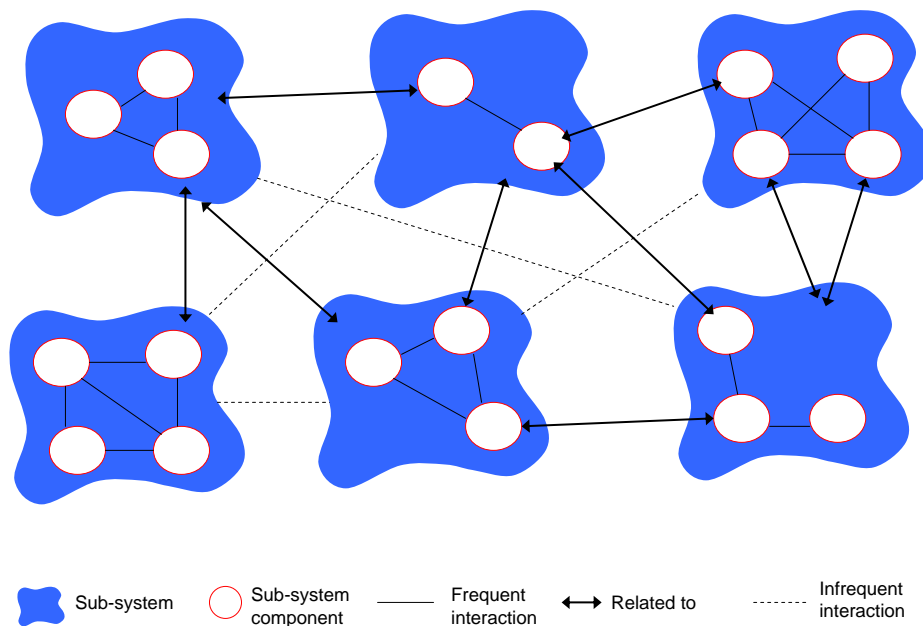


Figure 11.1: View of canonical complex system

- It is possible to distinguish between the interactions among subsystems and those within subsystems. The latter are both more frequent (typically at least an order of magnitude more) and more predictable than the former. This gives rise to the view that complex systems are nearly decomposable: subsystems can be treated almost as if they are independent, but not quite since there are some interactions between them. Moreover, although many of these interactions can be predicted at design time, some cannot.

Drawing these insights together, it is possible to define a canonical view of a complex system (Figure 11.1). The system's *hierarchical nature* is expressed through the “related to” links, components within a subsystem are connected through “frequent interaction” links, and interactions between components are expressed through “infrequent interaction” links. Given these observations, software engineers have devised a number of fundamental tools of the trade for helping to manage complexity [12]:

- *Decomposition*: The most basic technique for tackling large problems is to divide them into smaller, more manageable chunks, each of which can then be dealt with in relative isolation (note the nearly decomposable subsystems in Figure 11.1). Decomposition helps tackle complexity because it limits the designer’s scope.
- *Abstraction*: The process of defining a simplified model of the system that emphasises some of the details or properties, while suppressing others. Again, this works because it limits the designer’s scope of interest at a given time.

- *Organisation*: The process of defining and managing the interrelationships between the various problem-solving components (note the subsystem and interaction links of Figure 11.1). The ability to specify and enact organisational relationships helps designers tackle complexity by: enabling a number of basic components to be grouped together and treated as a higher-level unit of analysis, and providing a means of describing the high-level relationships between various units.

11.2 Complex Systems and Hierarchies

The goal of science is to understand and appreciate the Nature and to use that understanding to create materials, devices and software that enhance our lives. We can learn much from Nature. Certainly we can learn how she approaches problems. We can also learn how she solves specific problems sensing, repairing damage, creating mechanical strength. Beyond that, the study of organisms tells us what can be achieved, what problems can be solved, where we can set our goals in exploiting this understanding to benefit society.

Nature's approach to building complex structures and functions is hierarchical. This guides us in two ways. On a practical level, we can mimic this scheme by building our own structures hierarchically: combining simple molecules to fashion more complex ones, then combining those, repeatedly, until very complex structures with greatly enhanced properties emerge. At the conceptual level, Nature can be our guide in designing a multi-component research program to mirror this hierarchy. At the base of this program is the continuous studying of simple structures and phenomena—continuing because there is still much to learn, and because it provides the fundamental building blocks for the study of complexity. In most systems in Nature where we leave off the partitioning and what subsystems we take as elementary is somewhat arbitrary.

For example, one of the most recent results in evolutionary biology is that complex systems call for *layered*, hierarchical explanations. A first fundamental result is the so-called *theory of the hierarchies* [82]: in order to understand biological systems, and their evolution over time as well, several different *layers* has to be accounted for—from genes, to cells, up to organisms, species and higher taxa. While each layer is in some sense autonomous, and exhibit its own independent laws and dynamics, at the same time layers are organised in a hierarchy, each one strictly connected with the upper and lower levels, each parts of bigger totalities, each also wholeness composed of smaller parts. When observing / understanding / explaining a biological system, then, many different levels of abstraction can be adopted in which in the case of biological systems may correspond, for instance, to the gene, cell, organism, or species levels [59]—and provide different but equally meaningful views over the systems.

Simon defined a *hierarchy system* as a system composed of interrelated subsystems, each of the latter being in turn hierarchic in structure until we reach some lowest level of

elementary subsystem [197]. Note that this is only taken to imply a structural relationship and not necessarily a formal organisational hierarchy, in which subordinate subsystems each report to a “boss” at a higher level. Simon then went on to examine the timescale of evolution of complex systems and to argue that hierarchic systems will evolve more quickly than non-hierarchic systems of comparable size. This is illustrated with a parable of two watchmakers, Hora and Tempus [197]. Each makes watches composed of 1000 parts each, but while those of Tempus have to be assembled in one whole, Hora’s are made up of three levels of subassemblies of 10 elements each. So, Tempus has to make a complete assembly in one go, and it is assumed that if he is interrupted, the partially completed assembly will fall apart. Hence, for each interruption, he will lose more work, and he will take many more attempts to produce a complete assembly. Hora, on the other hand, has to complete 111 subassemblies for each complete watch, but she will lose less work for each interruption and will take far fewer attempts to make a complete assembly. If the probability of interruption is about 1 in 100, then Tempus will take around 4000 times as long as Hora to assemble a complete watch.

Simon argued that the same principle of faster evolution of a complex structure consisting of relatively stable substructures will apply to any biological or social system and so such hierarchic systems are likely to be much more common than non-hierarchic complex systems. For example, a problem-solving process, such as safe cracking, consisting of selective trial and error, in which partially successful approaches are retained, will find a solution much more rapidly than a completely random trial and error process.

Furthermore, many hierarchies form nearly decomposable systems, in which the interactions between subsystems are weak but not negligible compared to those within subsystems. In this case, the short-run behaviour of each subsystem may be analysed as approximately independent of the short-run behaviour of the other components, and the long-run behaviour similarly is seen to only depend in an aggregate way on the behaviour of the other components.

Thus, hierarchic complex systems are argued both to be more common and to be more comprehensible, often in terms of a process description of the dynamics of how they are constructed rather than a state description of their final configuration.

11.3 Complex Systems and Self-organisation

Self-organisation may be defined as a spontaneous (i.e. not steered or directed by an external system) process of organisation, i.e. of the development of an organised structure as a result of many local interactions. In other words, organisation occurs without any central organising structure or entity. Such self-organisation has been observed in systems at scales from neurons to ecosystems. The cooperative behaviour of self-organising systems results from local interactions between its members and not from the existence of a central controller is referred to as *emergent behaviour*. Swarms are one of the many self-

organising systems that are now being studied. The behaviour of such complex systems is typically unpredictable, yet exhibits various forms of adaptation and self-organisation. The idea that an ant colony is a system that organises itself without any leader is intriguing. Each individual ant, acting with limited information, contributes to the emergence of an organised whole. Another typical example is an ecosystem, consisting of organisms belonging to many different species, which compete or cooperate while interacting with their shared physical environment.

When we consider a highly organised system, we usually imagine some external or internal entity that is responsible for guiding, directing or controlling that organisation [90]. For example, most human organisations have a president, chief executive or board of directors that develops the policies and coordinates the different departments. Although the controlling entity (president) is part of the system, it is in principle possible to separate it from the rest. The controller is a physically distinct subsystem, that exerts its influence over the rest of the system. In this case, we may say that control is centralised. In self-organising systems, on the other hand, “control” of the organisation is typically distributed over the whole of the system. All parts contribute evenly to the resulting arrangement. Others interesting characteristics of self-organising systems are:

- *Robustness or resilience* — This means that self-organising systems are relatively insensitive to perturbations or errors, and have a strong capacity to restore themselves, unlike most human designed systems. For example, an ecosystem that has undergone severe damage, such as a fire, will in general recover relatively quickly. One reason for this fault-tolerance is the redundant, distributed organisation: the non-damaged regions can usually make up for the damaged ones. Another reason for this intrinsic robustness is that self-organisation thrives on randomness, fluctuations or “noise”; a third reason for resilience, the stabilising effect of feedback loops.
- *Non-linearity* — Most of the systems modelled by the traditional mathematical methods of physics are linear. This means basically that effects are proportional to their causes: if you kick a ball twice as hard, it will fly away twice as fast. In self-organising systems, on the other hand, the relation between cause and effect is much less straightforward: small causes can have large effects, and large causes can have small effects. This non-linearity can be understood from the relation of feedback that holds between the systems components. Each component affects the other components, but these components in turn affect the first component. Thus the cause-and-effect relation is *circular*: any change in the first component is fed back via its effects on the other components to the first component itself. Feedback can have two basic values: *positive* or *negative*. Feedback is said to be positive if the recurrent influence reinforces or amplifies the initial change. In other words, if a change takes place in a particular direction, the reaction being fed back takes place in that same direction. Feedback is negative if the reaction is opposite to the initial action, that is, if change is suppressed or counteracted, rather than

reinforced. Negative feedback stabilises the system, by bringing deviations back to their original state. Positive feedback, on the other hand, makes deviations grow in a runaway, explosive manner. It leads to accelerated development, resulting in a radically different configuration.

In the context of MAS, agents naturally play the role of autonomous entities subject to self-organise themselves. Usually agents are used for simulating self-organising systems, in order to better understand or establish models. The tendency is now to shift the role of agents from simulation to the development of distributed systems where components are software agents that once deployed in a given environment self-organise and work in a decentralised manner towards the realisation of a given (global) possibly emergent functionality. Researchers have been experimented with several mechanisms leading to self-organisation and often at the same time to emergent phenomenon on different kinds of applications. The different approaches can be divided in five classes depending on the mechanisms they are based [50]:

- *direct interactions*: the approaches proposed consist in using few basic principles, such as localisation and broadcast, coupled with local interactions and local computations done by agents in order to provide a final coherent global state. These mechanisms focus on changing the structural aspects of the agent organisation, such as topological placement of agents and agent communication lines.
- *indirect interactions and stigmergy*: the mechanisms aim at achieving complex system behaviours resulting of indirect interactions between agents. These interactions are due to changes in the environment. This behaviour leads towards the desired global system behaviour. In these cases, due to the non-linearity and the complexity of the phenomena involved, neither it is possible to have direct control of the system behaviour nor can it be proven that the desired behaviour will be achieved. The resulting system state cannot be accurately known in advance and multiple solutions can be reached. One can only obtain some statistical confidence about the system convergence to the desired globally behaviour with experimentation.
- *reinforcement*: these approaches are based on adaptive behaviour capabilities of individual agents which are dependent on particular agent architectures. Agents dynamically select a new behaviour based on the calculation of a probability value which is dependent on the current agent state and the perceived state of the environment, as well as on the quality of the previous adaptation decisions. It consists in the following basic principles: rewards increase agent behaviour and punishments decrease agent behaviour.
- *cooperation*: composition merges two agents into one and can be useful when communication overheads between the two agents are too high. The system tries to be

cooperative with its environment in creating one agent or in merging two agents in order to improve the response time to the environment. The initial organisation starts with one agent containing all domain and organisational knowledge. Simulation results demonstrate the effectiveness of the approach in adapting to changing environmental demands.

- *generic architecture*: a particular class of self-organisation mechanisms is based on generic reference architectures or meta-models of the agents' organisation which are instantiated and subsequently dynamically modified as needed according to the requirements of the particular application. A common aspect in reference architectures is that they involve characteristic agent types from which the basic agents of a holonic organisation are derived.

Self-organisation and emergence interest more and more the community of computer scientists and in particular the MAS developers. This craze is due to the fact that self-organisation enables to tackle a new field of applications and that multi-agent systems are well adapted to implement self-organisation.

11.4 Holonic Systems: Hierarchies and Self-organisation

The idea of hierarchy and of their constituent *part-wholes*, or *holons* [107], has a long and distinguished history. There are many philosophers who have proposed abstract systems for explaining natural and social phenomena. In pre-Socratic Greece Leucippus and Democritus developed the abstract concept of the atom and used it to develop a philosophy that could explain all observed events. Aristotle used hierarchy as the methodology for accumulating and connecting biological knowledge. Hierarchy was perhaps the dominant way of viewing the connection between the natural, the human and the supernatural orders of being through the middles ages.

The word holon is a combination of the Greek “holos” meaning whole, with the suffix “on” which, as in proton or neutron, suggests a particle or part. The holon, then, is a part-whole. It is a nodal point in a hierarchy that describes the relationship between entities that are self-complete wholes and entities that are seen to be other dependent parts. As one's point of focus moves up, down, and / or across the nodes of a hierarchical structure so one's perception of what is a whole and what is a part will also change [58].

Koestler noted that in every order of existence, from physical to chemical to biological and social systems, entirely self supporting, non-interacting entities did not exist [107]. And more importantly, that entities can be seen to lie in holarchical relationship with each other. Every identifiable unit of organisation, such as a single cell in an animal or a family unit in a society, comprises more basic units (mitochondria and nucleus, parents and siblings) while at the same time forming a part of a larger unit of organisation (a muscle tissue and organ, community and society). A holon, as Koestler devised the term,

is an identifiable part of a system that has a unique identity, yet is made up of subordinate parts and in turn is part of a larger whole. Koestler's holons were not thought of as entities or objects but as systematic ways of relating theoretical structures. In other words, holons were arbitrary points of reference for interpreting reality. Because holons are defined by the structure of a hierarchy each identified holon can itself be regarded as a series of nested sub-hierarchies in the same way that a set of Russian dolls is an inclusive series of dolls contained within each other [58]. Holons are, then, both parts and wholes because they are always parts of larger hierarchies and they always contain sub-hierarchies. Holons simultaneously are self-contained wholes to their subordinated parts, and dependent parts when seen from the inverse direction. Hence, holons can be seen as reference points in hierarchical series or holarchies. Koestler also recognised that holons are the representative stages or nodal structures that define the developmental hierarchies. As he says "the different levels represent different stages of development, and the holons reflect intermediary structures at these stages".

The strength of holonic organisation, or holarchy, is that it enables the construction of very complex systems that are nonetheless efficient in the use of resources, highly resilient to disturbance, and adaptable to changes in the environment in which they exist [78]. Moreover, holons may participate in multiple hierarchies at the same time. Holarchies are recursive in the sense that a holon may itself be an entire holarchy that acts as an autonomous and cooperative unit in the first holarchy. The stability of holons and holarchies stems from holons being self-reliant units, which have a degree of independence and handle circumstances and problems on their particular level of existence without asking higher level holons for assistance. The self-reliant characteristic ensures that holons are able to survive disturbance. The subordination to higher level holons ensures the effective operation of the larger whole.

Holons was initially successfully adopted in manufacturing system where they are viewed as intelligent, autonomous, and cooperative building blocks, within a manufacturing system, for transforming, transporting, storing and/or validating information and physical objects [116]. The direct linkage of holons with the physical elements of manufacturing and material handling infrastructure is stressed. A holon always contains an information processing part and, optionally, a physical processing part. The appearance and the whole existence of holons is tightly connected with the requirement of reconfigurability and holons are considered as the lowest level of granularity in the reconfiguration tasks. Cooperation among holons is supported by an evolutionary self-organising holarchy. The other important feature of holons is their recursiveness that means holons can contain another holons of the same architecture/ structure. Holons are able to communicate via information exchange, cooperate via participation in one or more cooperation domains in which they are able to perform a certain group-decision making exploring simple negotiations.

The holonic terminology is used mainly in the control-engineering domain. Several industrial holonic-based solutions were already presented, but special products supporting

a holonic-based automation for manufacturing and control are still missing on the market. The HMS (Holon Manufacturing Systems) consortium [91] did a significant progress in defining first standards for holonic systems. The basic holonic attributes defined by HMS consortium are [91]:

- **Autonomy:** each holon must be able to create, control and monitor the execution of its own plans and/or strategies, and to take suitable corrective actions against its own malfunctions.
- **Cooperation:** holon must be able to negotiate and execute mutually acceptable plans and take mutual action against malfunctions.
- **Openness:** the system must be able to accommodate the incorporation of new holons, the removal of existing holons, or modification of the functional capabilities of existing holons, with a minimal human intervention, where holons or their functions may be supplied by a variety of diverse sources.

Now it has been clearly recognised that intelligent solutions for engineering holonic systems can be achieved by encapsulating the function block solutions into a certain kind of higher-level software enabling more sophisticated, more intelligent negotiation process based on a richer knowledge representation schemas. Thus, the holonic research started to look for suitable agent-based solutions.

The progress achieved on both the areas of holonic and multi-agent systems as well as mutual convergence of these areas during the last year has been really significant. The researchers in these areas clearly identified the role of each of them in manufacturing and industrial control and recognised weak and strong points of each of these approaches [79]. Giret and Botti [78] compared holons and agents and the results are summarised in Figure 11.2

As it is possible to see in the figure, holon paradigm and agent paradigm share similar concepts: we can say that a holon is a special case of agent. At the lowest implementation level, both models are simple blocks of executable code with data flowing from / to them according to the particular implementation [78].

In MAS design, a holon constitutes a way to gather local and global, individual and collective points of view. A holon is thus a self-similar structure composed of holons as sub-structures and the hierarchical structure composed of holons is called a holarchy [37]. A holon can be seen, depending on the level of observation, either as an autonomous “atomic” entity or as an organisation of holons.

Two overlapping aspects have to be distinguished in holons: the first is directly related to the holonic nature of the entity (a holon, called “super-holon”, is composed of other holons, called sub-holons or members) and deals with the government and the administration of a super-holon. This aspect is common to every holon and thus called the holonic aspect. The second aspect is related to the problem to solve and the work to be done. It

<i>Property</i>	<i>Holon</i>	<i>Agent</i>
Autonomy	Yes	Yes
Reactivity	Yes	Yes
Pro-activity	Yes	Yes
Social Ability	Yes. Human Interface is specific to of each holon.	Yes. Human Interface is generally implemented by one or several specialized agents.
Cooperation	Yes. Holons never deliberately reject cooperation with another holon.	Yes. It may compete and cooperate.
Organization, openness	Yes. Holarchies.	Yes. Hierarchies, horizontal organizations, heterarchies, etc. Holarchies can be implemented using several MAS architecture approaches for federations such as facilitators, brokers, or mediators.
Rationality	Yes	Yes
Learning	Yes	Yes
Benevolence	Yes	Yes
Mobility	Holons rarely need mobility for the execution of their tasks.	Yes
Recursiveness	Yes	There is no recursive architecture as such, but some techniques could be used to define federations that could simulate different recursive levels.
Information processing and physical processing	Yes. The separation is explicit, although the Physical Processing part is optional.	There is no explicit separation.
Mental attitudes	Yes. They do not need to reason on their own mental attitudes or those of other control units.	Yes

Figure 11.2: Holons vs. Agents [78]

depends on the application or application domain. It is therefore called the production aspect.

11.5 Summing up

This chapter has introduced the complex systems and their two key features: hierarchies and self-organisation. Although there is no universally accepted definition of a complex system, most researchers would describe as complex a system of connected entities that exhibits an emergent global behaviour resulting from the interactions between the entities. As shown in this chapter the agent paradigm seems very suitable for modelling and simulating this kind of systems. Traditional software engineering techniques are well-suited for capturing the hierarchical system organisations but they are insufficient for capturing self-organising aspects, since they are based on interfaces fixed at design time, or well established ontologies. As for current methodologies, they only make it possible to define a global behaviour when it is a mere sum of the behaviour of the various parts. Current practices in multi-agent systems directly address self-organisation and consist in designing distributed algorithms taking inspiration from natural mechanisms both bio-inspired and socially-inspired. Some agent-oriented methodologies such as ADELFE provide to the designer means to design self-organising systems [50]. However the whole engineering process underpinning methodologies remains open issue.

12

Managing System Complexity

As highlighted in the previous chapter, complexity is inherent in real-life systems [53]. While *modelling* complex systems and understanding their behaviour and dynamics is the most relevant concern in many areas, such as economics, biology, or social sciences, in the software systems also the complexity of *construction* becomes an interesting challenge. An integral part of a system development methodology must therefore be a set of tools for controlling and managing this complexity. So this chapter presents the tools that support the complexity of the system representations. In particular this chapter focus on those tools for managing the hierarchical aspects of complex systems. Even if self-organisation and emergent behaviour are very important characteristics of complex system they are not considered in this chapter because they are out of the scope of this thesis.

The very need for systems analysis and design strategies stems from complexity. If systems or problems were simple enough for humans to be grasped by merely glancing at them, no methodology would have been required. Due to the need for tackling sizeable, complex problems, a system development methodology must be equipped with a comprehensive approach, backed by set of reliable and useful tools, for controlling and managing this complexity. Like most classical engineering problems, complexity management entails a tradeoff that must be balanced between two conflicting requirements: *completeness* and *clarity*. On one hand, completeness requires that the system details be stipulated to the fullest extent possible—the system must be specified to the last relevant detail. On the other hand, the need for clarity imposes an upper limit on the level of complexity of each individual diagram and does not allow for a diagram that is too cluttered or loaded—the documentation must be legible and comprehensible. To tackle complex systems, a methodology must be equipped with adequate tools for complexity management that address and solve this problem of completeness-clarity tradeoff by striking the right balance between these two contradicting demands. The theories and tools adopted in the managing of complex system (Chapter 11) – in particular decomposition, abstraction and layering – could be very useful in the formulation of a new approach for managing the complexity in the system representation.

The remainder of this chapter is organised as follows: Section 12.1 presents the middle-out approach for system's analysis and design, Section 12.2 and Section 12.3 illustrate

respectively as the object-oriented and AO notations and methodologies manage the complexity of representation. Section 12.4 propose a new mechanism for managing the complexity in AO methodologies. Finally Section 12.5 presents a summary of the chapter.

12.1 Middle-out as the de-facto Practice

Analysing is the process of gradually increasing the human analyser’s knowledge about the system’s structure and behaviour. Designing is the process of gradually increasing the amount of detail on the system’s architecture, i.e., the structure and behaviour combination that enables the system to attain its function. For both analysis and design, managing the system’s complexity therefore entails being able to present and view the system at various levels of detail that are consistent with each other [53]. Ideally, analysis and design start at the top and make their way gradually to the bottom—from the general to the detailed. In real life, however, analysis typically starts at some arbitrary detail level and it is rarely linear [128]. The design is not linear either. Usually these are iterative processes, during which knowledge, followed by understanding, is accumulated and refined by degrees.

The system architect usually cannot know in advance the precise structure and behaviour of the very top of the system—this requires analysis and becomes apparent at some point along the analysis process. Step by step, the analyst builds the system specification by accumulating and recording facts and observations about things in the system and relations among them. The sheer amount of detail contained in any real world system of reasonable size overwhelms the system architects soon enough during the architecting process. Trying to incorporate the details into one diagram and their interconnections quickly becomes an entangled web. This information overload happens even if the method advocates using multiple diagram types for the various system aspects. Because the diagram has become so cluttered, it is increasingly difficult to comprehend it. System architects experience this detail explosion phenomenon on a daily basis, and anyone who has tried to analyse a system will endorse this description. The problem calls for effective and efficient tools to manage this inherent complexity.

Due to the non-linear nature of these processes, linear, unidirectional “bottom-up” or “top-down” approaches are rarely applicable to real-world systems. Rather, it is frequently the case that the system under construction or investigation is so complex and unexplored, that neither its top nor its bottom is known with certainty from the outset. More commonly, analysis and design of real-life systems start in an unknown place along the system’s detail level hierarchy. The analysis proceeds *middle-out* by combining top-down and bottom-up techniques to obtain a complete comprehension and specification of the system at all the detail levels. It turns out that even though the architect usually strives to work in an orderly top-down fashion, more often than not, the de-facto practice is the middle-out mode of analysis and design.

During the middle-out analysis and design, facts and ideas about objects in the system and its environment, and processes that transform them, are being gathered and recorded. As the development proceeds, the system architect tries to concurrently specify both the structure and the behaviour of the system in order to enable it to fulfill its function. For an investigate system, the researcher tries to make sense of a long list of gathered observations and to understand their cause and effect relations. In both cases, the system's structure and behaviour go hand in hand, and it is very difficult to understand one without the other.

12.2 System Complexity in OO Methodologies and Notations

Rooted in military art, the *decomposition principle*, also known as the *divide and conquer strategy*, has been recognised for a long time and in many domains as an effective means to overcome complexity and enable the solving of complex problems. The idea is basically to break a complex problem (such as understanding and/or designing a complex system) into smaller, manageable pieces, solve each of them separately and combine the partial solutions to obtain a complete solution. Obviously this principle is not enough for managing the complexity in those systems where the system behaviour *emerge* from the local interactions at run-time (see Section 11.3). These systems need different tools that are out of the scope of this work.

System development methodologies have adopted the decomposition principle, either intentionally or not. Most methodologies apply this strategy by breaking the system into a number of models, each dealing with a different aspect of the system, such as structure, behaviour and function. Each model applies a different set of symbols and concepts, and together they are expected to convey a complete system specification. This *aspect decomposition* is at the heart of standard, state-of-the-art object-oriented development languages like UML. A system is then expressed as a multiplicity of different models, each representing a specific system aspect: actually, UML defines thirteen types of diagrams, four of which represent the static application structure, five are devoted to capture the system's dynamic behaviour, and three are related to the organisation and management of application modules [123]. Altogether, all these models are expected to convey a complete system specification.

However, although the availability of so many models constitutes a richness from the expressiveness viewpoint, each model introduces its own set of symbols and concepts, thus leading to an unnatural complexity in terms of vocabulary, model multiplicity and model integration [54]. This is a problem both for maintaining consistency among the different system models and views, and for the mental integration of such views, since integrating several models within one's mind is a very difficult process. That is why the need to concurrently refer to different models in order to understand a system and the way it

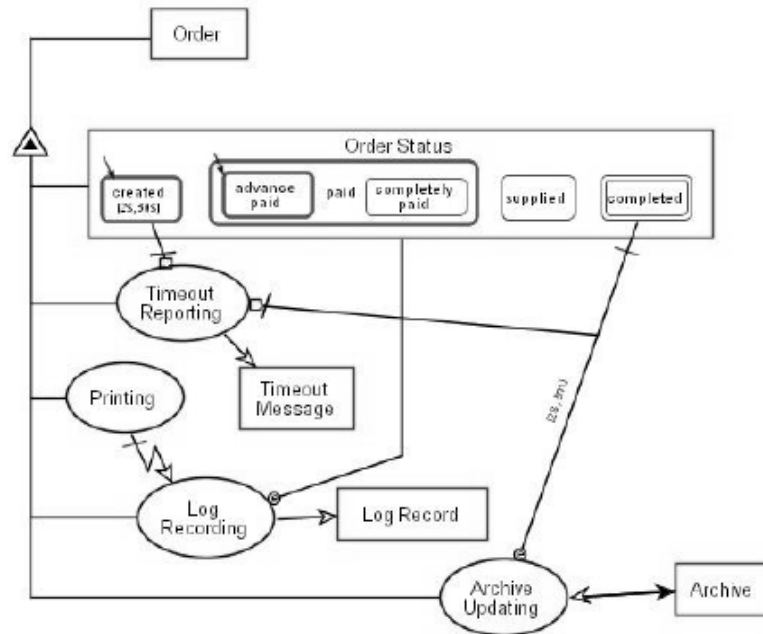


Figure 12.1: Example of folding in an object process diagram [54]

operates and changes over time is a critical issue, known as the *multiplicity problem* [171].

A different notation, OPM, adopts a different approach. A basic principle in OPM is that structure and behaviour within a system are so intertwined that effectively separating them is extremely harmful, if not impossible. OPM has adopted the *detailed decomposition*: rather than decomposing a system according to its various aspects, the decomposition is based on the system's levels of abstraction. OPM controls complexity through granularity levels, refinement, and abstraction similar to zooming in and zooming out in a digital map. Three built-in refinement/abstraction mechanisms are built into OPM. They enable presenting the system elements at various detail levels without losing the comprehension of the system as a whole.

12.2.1 Detail Decomposition in OPM

OPM controls the complexity through granularity levels, refinement, and abstraction similar to zooming in and zooming out in a digital map. There are three *scaling modes* with respect to the three entities of OPM—object, process and state. *Visibility scaling* zooms in or zooms out of a process, *hierarchy scaling* unfolds or folds an object, and *manifestation scaling* expresses or suppresses a state. These are done via three *refinement/abstraction mechanisms*: *unfolding/folding*, *in-zooming/out-zooming*, and *state expressing/suppressing*.

Unfolding/folding is applied by default to objects for detailing/hiding their structural

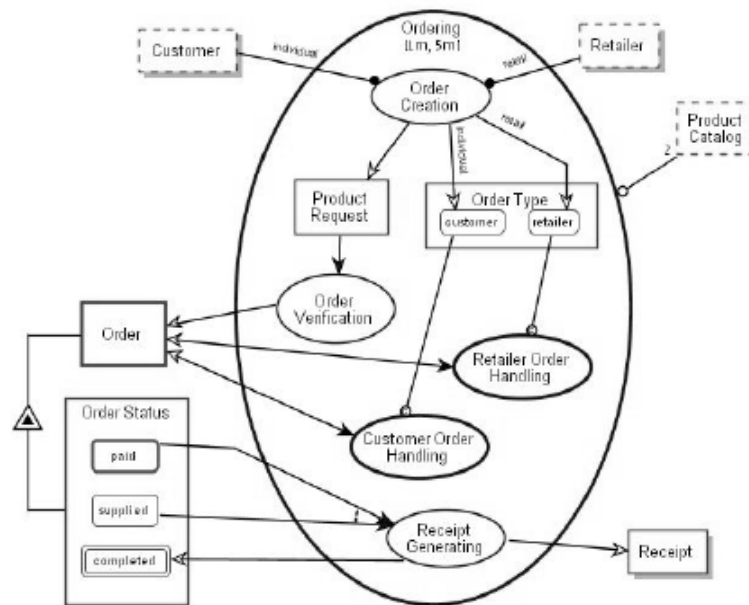


Figure 12.2: Example of in-zooming in an object process diagram [54]

components (parts, specialisations, features, or instances). Unfolding reveals a set of lower-level entities that are hierarchically below a relatively higher-level thing. The hierarchy is with respect to one or more structural links. The result of unfolding is a graph, the root of which is the thing being unfolded. Linked to the graph are the things that are exposed as a result of the unfolding. Conversely, folding is applied to a tree from which a set of unfolded entities is removed, leaving just the root. Figure 12.1 shows an example in which the Order object is unfolded, showing its operations and event triggers. Unfolding/folding can be applied fully or partially to any subset of descendants (parts, specialisations, features, or instances) of a thing (object or process).

A state is a situation in which an object can be for some period of time. At any point in time an object is in exactly one of its states. State expressing is a refinement mechanism applied to objects which reveals a set of states inside an object. State suppressing is the abstraction mechanism which conceals a set of states inside an object. Figure 12.1 shows an example in which Order Status is expressed.

In-/out-zooming is applied by default to processes for detailing/hiding their sub-process components and details of the process execution. In-zooming of (i.e., zooming into) an entity decreases the distance of viewing it, such that lower-level elements enclosed within the entity become visible. Conversely, out-zooming (i.e., zooming out) of a refined entity increases the distance of viewing it, such that all the lower-level elements that are enclosed within it become invisible. Figure 12.2 shows an example of the in-zooming of the Ordering process.

Despite the different names, all the OPM scaling mechanisms allow engineers to work middle-out: MAS engineers can choose to start at any arbitrary abstraction level, and then achieve both the most detailed level and the most abstract level, along with the entire spectrum of intermediate levels between these two extremes.

12.3 System Complexity in AO Methodologies

As advocated in [158], MASs, once developed up to their full potential, can be generally seen as representing a class of complex artificial systems, wide and meaningful enough to legitimise, in principle, the application to MASs of the general principles and laws governing complex systems. This means that it is possible to apply the “hierarchy principle” (see Section 11.2) to the design of MASs: this first suggests that MAS models, abstractions, patterns and technologies can be suitably categorised and compared using a *layered description*. More simply and directly, when applied to the engineering of MASs, the hierarchy principle suggests that agent-oriented processes and methodologies should support some forms of MAS layering, allowing engineers to design and develop MAS along different levels of abstractions—a number of independent, but strictly related, MAS layers (Chapter 11).

Accordingly, one should expect that existing methodologies actually do support abstractions and processes for MAS layering—like for example the OPM’s zooming. Quite interestingly, however, current AO methodologies offer very little (if any) support for hierarchical representation of MASs. So, in the following subsection the main AO methodologies are surveyed to look for some support for layered representation of MAS.

12.3.1 AO Methodologies & Layering

Many methodologies exist in the literature aimed at the engineering of artificial systems in terms of MASs (Chapter 4). Although none of those methodologies provides MAS engineers with an explicit layering mechanism, some of them exhibit some implicit mechanisms that make it possible in some sense to analyse the system at different levels of detail.

To the best of our knowledge, the most cited AO methodology, Gaia, does not introduce any mechanism providing for MAS layering. In MaSE, instead, two models allow MASs to be represented at different levels of abstraction: the *creating-agent-classes* model should provide a high-level vision of the MAS agents and of their main conversations; the *assembling-agent-classes* model “zooms” on the inner agent structure, and provides for a number of predefined components, which may also have sub-architectures (with further sub-components) of their own.

Tropos promotes a form of refinement across different stages of the MAS analysis process, such as when the *actor* and *dependency* models built in the *early requirements*

phase are extended during the *late requirements* phase by adding the system-to-be as another actor, along with its inter-dependencies with social actors. Also MESSAGE [69] uses a refinement model in the analysis phase: the level 0 model gives an overall view of the system, its environment, and its global functionality; next level 1 defines the structure and the behaviour of entities such as organisation, agents, tasks, goals, domain entities; further levels (2, 3, ...) might be defined for pointing out specific aspects of the system dealing with functional requirements, as well as non-functional requirements such as performance, distribution, fault tolerance, security. In Prometheus [164], a progressive refinement process is used which starts by describing agents internally in terms of capabilities. The internal structure of each capability is then given, optionally using or introducing further capabilities, which are refined in turn until all capabilities have been defined: capability nesting is allowed, thus allowing for arbitrarily many layers, in order to achieve an understandable complexity at each level.

Finally a recent proposed methodology called ASPECS [37] suggests a method for managing the complexity of the representation tied to the system holonic structure (Section 11.4) where holons can be seen, depending on the level of observation, either as an autonomous “atomic” entity or as an organisation of holons. The main vocation of ASPECS is towards the development of societies (organisations) of holonic (as well as non-holonic) multi-agent systems. ASPECS proposes a *multiview approach* that consists in merging the various points of view on the system (including the functional and ontological views). This approach respects the hierarchical nature of the system revealed by the ontological approach – where the system ontology is hierarchical decomposed, and Uses cases referring to ontological concepts of the same level are grouped – but it clearly separates use cases attached to different system functionalities. The granularity of functionalities and the different levels of abstraction present in the system are respected. The hierarchical decomposition of ontology inducts a decomposition of the global behaviour embodied by an organisation into smaller interacting behaviours. Each organisation is decomposed and it is specified the precise behavioural contribution of sub-level organisations to an upper-level organisation.

The above forms of layering, however, are quite limited and more tied to the structure of the system – ASPECS in particular – than to view the system at different level of abstractions. First of all, they enforce only a top-down, mono-directional form of zooming—so, refinement is allowed, abstraction is not allowed. Then, they have only a pre-fixed scope and structure, which limits in principle their flexibility and possibly their ability to fit the many different MAS application scenarios. Mechanisms for zooming are then not explicit, and no ontological support is currently provided by any of the available AO methodologies to the best of our knowledge.

So, it is no surprise that OPM was extended in order to support concepts from the agent field. OPM/MAS [201] takes MAS building-blocks from the Gaia methodology. The set of MAS building blocks is divided into two groups: the first contains static, declarative building blocks, while the second group contains building blocks with a behavioural,

dynamic nature. The building blocks in the first group – which include organisation, society, platform, rule, role, user, protocol, belief, desire, fact, goal, intention, and service – are modelled as OPM objects; the building blocks in the second group – which include agent, task, and messaging – are modelled using the process concept.

OPM/MAS is indeed the first actual effort to introduce the zooming mechanism into a methodology for modelling multi-agent systems. However, apart from the obvious problems arising from the uneven mixture of the very different OPM and Gaia approaches, OPM zooming mechanisms appear too generic for Gaia-derived agent abstractions, which were not conceived with zooming in mind. For instance, in-zooming an agent according to OPM rules would generally lead to another object or to another agent, more plausibly: there is no way that an activity represented at a given abstraction level as an agent could become a society of agents at the next, more refined level.

12.4 Layering Mechanisms for AO Methodologies: A First Insight

Taking inspiration from the OPM's in-zooming/out-zooming and from the hierarchical structure of complex system (Section 11.2), a simple layering principle with the specific aim of scaling with the complexity of system description can be added to AO methodologies. Different from OPM that provides other two different mechanisms – unfolding/folding and expressing/suppressing – for managing the complexity, the proposed principle adopts only one mechanism – zooming – in order to simplify the layering managing. Even if the OPM approach seems very easy but if we carefully analyse the OPM meta-model related to the scaling process [53] it appears not so easy: we can note that this process presents different attributes that specifies the kind of scaling and the relative default entities subject to the scaling operation and so on. Indeed, our purpose is to have a mechanism applicable in a uniform way at all the abstractions supported by methodologies.

The layering principle comes from the basic intuition that what can be described as a complex task or goal assigned to a role R at a generic layer L , can be zoomed into a different subtasks or subgoals assigned to a group of roles at the layer $L+1$ —and vice versa. Each layer contains a description of the models at a given level of abstraction, and is labelled with a number: as a convention, the uppermost layer is the layer 0—which represents the most abstract view of the MAS: so, zooming a model at layer L results in a model at either layer $L+1$ (in-zooming) or layer $L-1$ (out-zooming) [128]. That is, zooming allows for different viewpoints over the system at different levels of abstraction (see Figure 12.3). The zooming mechanism provided for the models at one stage directly impacts on the models and diagrams identified at all the other stages. For example in Figure 12.3 the effect of the application of the in-zoom at a model in one stage inducts in the subsequently models an in-zoom operation: the agent mapping a role at the layer

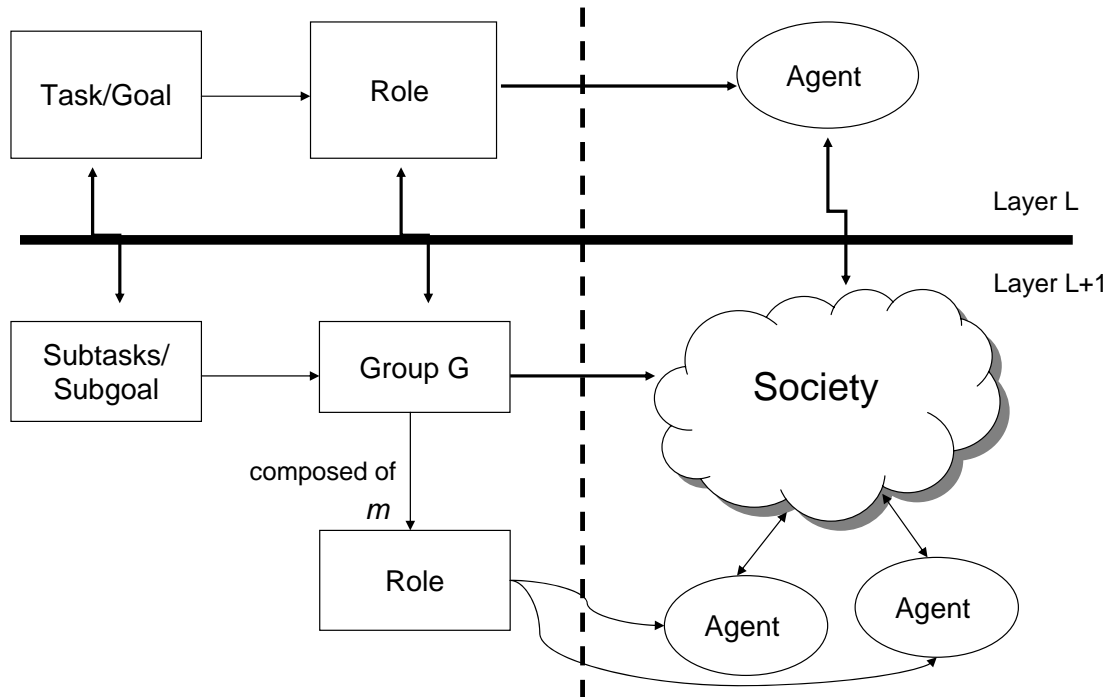


Figure 12.3: Zooming: the basic intuition

L is in-zoomed into a society of agents at the layer L+1, mapping the group which results from in-zooming R. Dually, as in-zoom allows for more and more detailed views over the systems, out-zooming provides engineers with a mechanism for abstraction. For example, a social task ST assigned to a group at layer L could be abstracted (out-zoomed) into an individual task IT assigned to an individual role R at (more abstract) layer L-1, thus concealing the roles of the group. In this way, we allow engineers to provide a more concise description of MASs, where the unnecessary details that could hinder system understanding are abstracted away, and only the main entities and their mutual relationships are actually accounted for. The availability of symmetric and uniform in-/out-zooming mechanisms promote middle-out approaches to the engineering of MASs.

12.4.1 Zoom & Artifacts

As mentioned in Chapters 7 and 9 artifacts have a relevant impact on the way in which any AO methodology could be organised.

What might not be so obvious is that the very notion of artifact is itself affected by the principles of the methodology, as it happens when they are introduced in a methodology. This is particularly evident when artifacts are introduced in a methodology that supports the layering principle [127](Chapter 14). For instance, an environmental artifact at layer L could be zoomed and become an aggregation of one or more social artifacts (managing

the access policy) and one or more environmental artifacts at layer $L+1$ (Figure 12.4).

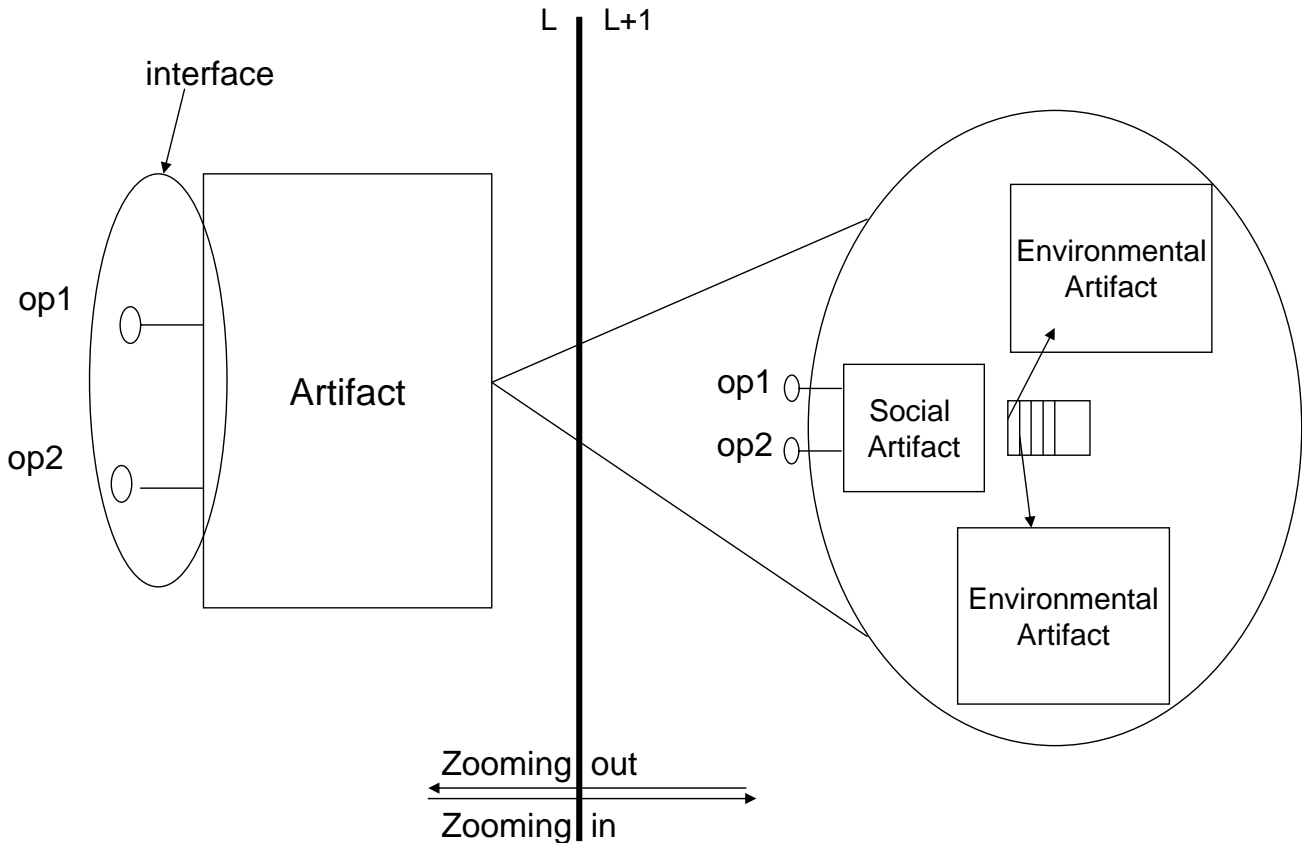


Figure 12.4: Zooming: exploding / imploding artifacts

Zooming artifacts also allows for different levels of abstraction over resources. As a simple example, taken from the real world, one may think of a simple desktop computer as an artifact: at layer L , it may be seen as a single resource artifact, but would become a composition of different environmental artifacts (a CPU, a hard-disk, a DVD unit, wires, and so on) when zoomed at layer $L+1$. If we further zoom in the hard-disk, this could be seen at layer $L+2$ as the composition of a case, disks, heads—just to mention a few; and zooming could continue until the desired / required level of detail / abstraction is reached.

However, zooming artifacts is not restricted only to “exploded” artifacts (Figure 12.4—i.e., an artifact that “generates” several artifacts. Instead, it could involve a *refinement* of the artifact’s features (Figure 12.5), such as its usage interface. So, an artifact could expose at layer L an interface that provides all the operations required at that layer, whereas at layer $L+1$ the same artifact could expose further, more detailed operations according to the level of abstraction required. For instance, an individual artifact operation could

result in a number of (more refined) artifact operations at layer $L+1$.

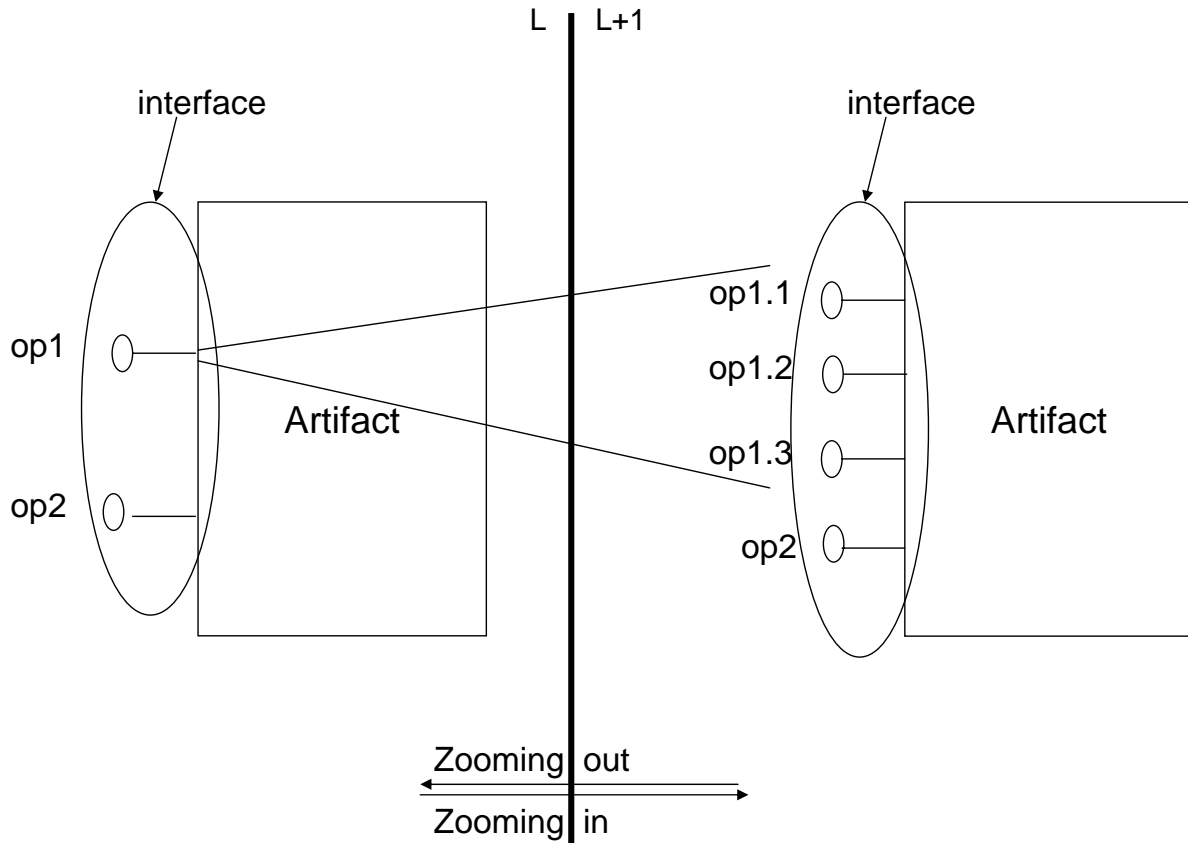


Figure 12.5: Zooming: refining artifacts

12.5 Summing up

This chapter has presented tools for managing the hierarchical aspects of complex systems both in the object-oriented and agent-oriented methodologies. In addition, this chapter has proposed a simple yet expressive layering principle for managing the complexity in AO methodologies. This principle takes inspiration from both the scaling mechanisms of OPM and the theory of the hierarchies (Section 11.2).

In the traditional literature, other object-oriented methodologies are defined that support some sort of layering: among these, the Booch method [83], EROOS [60] and OSA [162]. Both Booch method and EROOS define iterative processes: phases are often repeated, each time focusing on a more detailed level of abstraction. OSA is somehow similar to our principle in the uniform, bidirectional application of its zooming mechanism, as "... High-Level Object Classes have exploded and imploded views. An exploded view shows what a High-Level Object Class contains, while the imploded view hides its

contents ...”. However, how OSA mechanism could provide for recursion (to allow for an unlimited number of abstraction layers), and how it could be generalised to be applicable to anything else than Object Classes is frankly unclear from the available literature.

So, it is now clear that any methodology aimed at engineering complex artificial systems should provide engineers with some tool allowing for expressive and consistent hierarchical descriptions of systems.

Part V

SODA

13

SODA: The Early Version

This chapter presents the first version of the **SODA** (Societies in Open and Distributed Agent spaces) methodology created by Andrea Omicini in 2000 [144]. This version represents the starting point of my Ph.D work that has led to a new formulation of **SODA** presented in the next chapters.

SODA in its first version is a methodology for the analysis and design of multi-agent systems. The goal of **SODA** is to define a coherent conceptual framework and a comprehensive software engineering procedure that accounts for the analysis and design of individual agents, agent societies, and agent environments.

The analysis phase is characterised by three models: the *role model*, the *resource model* and the *interaction model*. The *design* phase is based on three strictly-related models, deriving from the models defined in the analysis phase; in particular, the analysis' role model maps to the design's *agent model* and *society model*, while the analysis' resource model maps to the design's *environment model*. The analysis' interaction model, in its turn, generates the interaction protocols and coordination rules referenced by the design's models.

SODA is not concerned with *intra-agent* issues: designing a multi-agent system with **SODA** leads to defining agents in terms of their required observable behaviour and their role in the multi-agent system. Then, whichever methodology one may choose to define the agent structure and inner functionality, it could be easily used in conjunction with **SODA**. Instead, **SODA** concentrated on *inter-agent* issues, like the engineering of societies and infrastructures for multi-agent systems. Since this conceptually covers all the interactions within an agent system, the design phase of **SODA** deeply relies on the notion of coordination model [165, 166]. Coordination models and languages were taken as the sources of the abstractions and mechanisms required to engineer agent societies: social rules were designed as coordination laws and embedded into coordination media, and social infrastructures were built upon coordination systems.

In the first formulation, **SODA** does not adopt a specific language notation for supporting the designers' work. This is because the available notations – such as UML [143] or its extension for agents AUML [64] – were not suitable for agent modelling and the **SODA** author did not create a new specific notation. However this choice is not so good

because the methodology is not actually usable: a designer that would use **SODA** should first create an ad hoc representation of the models and then integrate this representation with the **SODA** process. On one hand this gives great freedom to the designers for adopting the notation language they prefer, on the other hand this does not allow the creation of a set of case studies developed with **SODA** that help the new users to understand the methodology.

This lack has been resolved by the introduction of a tabular representation [128], which describes and relates **SODA**'s entities by means of tables. This formalism has been chosen for its simplicity and clearness: the designers are still free to adopt a specific language for the entity description ranging from an informal textual description to more formal languages; however now there are guidelines for the organisation of the entities, their related features and their relationships.

Therefore, in the following first the analysis phase and the design phase – along with the tabular representation – are respectively illustrated in Section 13.1 and Section 13.2, the **SODA** meta-models are presented (Section 13.3), and finally a discussion about the limitation of this version is reported in Section 13.4.

13.1 The Analysis Phase

During the analysis phase, the application domain is studied and modelled, the available computational resources and the technological constraints are listed, the fundamental application goals and targets are pointed out.

The result of the analysis phase is typically expressed in terms of high-level abstractions and their mutual relationships, providing designers with a formal or semi-formal description of the intended overall application structure and organisation. Since by definition agents have goals that they pursue pro-actively, agent-oriented analysis can rely on agent responsibility to carry on one or more tasks. Furthermore, agents live in an environment, which may be distributed, heterogeneous, dynamic, and unpredictable. So, the analysis phase should explicitly take into account and model the required and desired features of the agent application environment, by modelling it in terms of the required resources and the services made available to agents. Finally, since agents are basically interactive entities, which depend on other agents and available resources to pursue their tasks, the analysis phase should explicitly model the interaction protocols in terms of the information required and provided by agents and resources.

So, the **SODA** analysis phase exploits three different models:

- the *role model* — the application goals are modelled in terms of the tasks to be achieved, which are associated to roles and groups (Subsection 13.1.1)
- the *resource model* — the application environment is modelled in terms of the services available, which are associated to abstract resources (Subsection 13.1.2)

Role	Task	Interaction Protocol
<i>role name</i>	<i>task name</i>	<i>list of protocols</i>

Figure 13.1: Role Table

- the *interaction model* — the interaction involving roles, groups and resources is modelled in terms of interaction protocols, expressed as information required and provided by roles and resources, and interaction rules, governing interaction within groups (Subsection 13.1.3)

The above models represent the basis of the **SODA** analysis phase. Even though conceptually distinct, they are obviously strictly related, and should be defined in a consistent way.

13.1.1 The Role Model

Tasks are expressed in terms of the responsibilities they involve, of the competences they require, and of the resources they depend upon. Responsibilities are expressed in terms of the state(s) of the world that should result from the task accomplishment. Tasks are classified as either *individual* or *social* ones. Typically, social tasks are those that require a number of different competences, and access to several different resources, whereas individual ones are more likely to require well-delimited competence and limited resources. Each individual task is associated to an *individual role*, which by consequence is first defined in terms of the responsibilities it carries. Analogously, social tasks are assigned to *groups*. Groups are defined in terms of both the responsibility related to their social task, and the social roles participating in the group. A social role describes the role played by an individual within a group, and may either coincide with an already defined (individual) role, or be defined ex-novo, in the same form as an individual one, by specifying its task as a sub-task of its groups one.

The role model can be represented by defining a *Role Table* (Figure 13.1) and a *Group Table* (Figure 13.2), respectively [128]. In addition, since **SODA** associates *interaction protocols* to roles and *interaction rules* to groups, one extra column is added to such tables, to represent these associations. Each group is associated to a set of roles: correspondingly, further Role Tables can be introduced to express these relationships—one (social) Role Table for each group. As a result, Role Tables are exploited to express both the individual roles (one table) and the social roles (as many tables as the groups are). Of course, the interaction protocols associated to roles and the interaction rules associated to groups are further detailed in the interaction model (Figures 13.4 and 13.5 below).

Group	Social Task	Interaction Rule
<i>group name</i>	<i>social task name</i>	<i>list of rules</i>

Figure 13.2: Group Table

13.1.2 The Resource Model

Services express functionalities provided by the agent environment to a multi-agent system—like recording information, querying a sensor, verifying an identity. In this phase, each service is associated to an abstract *resource*, which is then firstly defined in terms of the services it provides. Each resource defines abstract access modes, modelling the different ways (*policies*) in which the services it provides can be exploited by roles. If a task assigned to a role or a group requires a given service, the access modes are determined and expressed in terms of the granted permission to access the resource in charge of that service. Such a permission is then associated to that role or group.

The *Resource Table* (Figure 13.3) expresses this relationship, also listing the interaction protocols associated to each resource (again, details about interaction protocols and rules are provided in the interaction model—Figures 13.4 and 13.5).

Resource	Service	Policy	Interaction Protocol
<i>resource name</i>	<i>list of services</i>	<i>list of permissions</i>	<i>list of protocols</i>

Figure 13.3: Resource Table

13.1.3 The Interaction Model

Analysing the interaction model in **SODA** amounts to the definition of *interaction protocols* for roles and resources, and *interaction rules* for groups. An interaction protocol associated to a role is defined in terms of the information required and provided by the role in order to accomplish its individual task. An interaction protocol associated to a resource is defined in terms of the information required to invoke the service provided by the resource itself, and by the information returned when the invoked service has been brought to an end, either successfully or not.

An interaction rule is instead associated to a group, and governs the interactions among social roles and among social roles and resources so as to make the group accomplish its social task.

It is worth noting that this approach ensures a form of uncoupling: each interaction protocol is not specifically bound to any other, and can be defined somehow

Interaction Protocol	Information Required	Information Provided
<i>name of protocol</i>		

Figure 13.4: Interaction Protocols Table

Interaction rule	Rule description
<i>name of rule</i>	<i>description</i>

Figure 13.5: Interaction Rules Table

independently—by simply requiring the specification of the information needed, but not its source: the same interaction protocol could potentially be exploited by different roles. Obviously, the final outcome of the analysis phase should account for this, too, by ensuring that for any information required by any protocol, there is at least one entity in the system in charge of supplying such information.

The interaction model can be represented by defining *Interaction Protocol Table* (Figure 13.4) and an *Interaction Rule Table* (Figure 13.5)

13.1.4 Analysis: the outcome

The results of the SODA analysis phase are expressed in terms of roles, groups, and resources.

To summarise:

- a role is defined in terms of its individual task and the corresponding interaction protocols,
- a group is defined in terms of its social task, the participating social roles, and the corresponding interaction rules,
- a resource is defined in terms of the service it provides, its access modes, the permissions granted to roles and groups to exploit its service, and the corresponding interaction protocol,
- an interaction protocol is defined in terms of information provided and information required,
- an interaction rule is defined by means of a textual description.

13.2 The Design Phase

Design is concerned with the representation of the abstract models resulting from the analysis phase in terms of the design abstractions provided by the methodology [144].

Agent	Role	Interaction Protocol	Resource	Permission
<i>agent name</i>	<i>role name</i>	<i>list of protocols</i>	<i>list of resources</i>	<i>list of permissions</i>

Figure 13.6: Agent Table

Different from the analysis phase, a satisfactory result of the design phases is typically expressed in terms of abstractions that can be mapped one-to-one onto the actual components of the deployed system.

The **SODA** design phase is based on three strictly related models:

- the *agent model* — individual and social roles are mapped to agent classes (Subsection 13.2.1)
- the *society model* — groups are mapped onto societies of agents, which are designed and organised around coordination abstractions (Subsection 13.2.2)
- the *environment model* — resources are mapped onto infrastructure classes, and associated to topological abstractions (Subsection 13.2.3)

13.2.1 The Agent Model

An *agent class* is defined as a set of (one or more) roles, both individual and social ones. As a result, an agent class is first characterised by the tasks, the set of the permissions, and the interaction protocols associated to its roles.

The design of the agents of a class should account for all the specifications coming from the **SODA** analysis phase but may exploit in principle any other methodology for the design of individual agents, since this issue is not covered by **SODA**. What is determined by **SODA** is the outcome of this phase, that is, the observable behaviour of the agent in terms of all its interactions with the surrounding environment. Such a behaviour is defined by the interaction protocols, delimited by the permission sets, and finalised to the achievement of the agent tasks.

The agent model is described by means of the *Agent Table* (Figure 13.6). Again, the interaction protocols referenced here are those defined in the Figure 13.4 of the analysis phase.

13.2.2 The Society Model

Each group is mapped onto a *society* of agents. So, an agent society is first characterised by the social tasks, the set of the permissions, the participating social roles, and the interaction rules associated to its groups. The agent model also assigns social roles to

Society	Group	Coordination Medium	Resource	Coordination Rule
<i>society name</i>	<i>group name</i>	<i>medium name</i>	<i>list of resources</i>	<i>list of rules</i>

Figure 13.7: Society Table

agents, so that the main issue in the society model is how to design interaction rules so as to make societies to accomplish their social tasks. Since it deals with managing agent interaction, the problem of achieving the desired social behaviour by means of suitable social rules is basically a coordination issue. As a result, societies in **SODA** are designed around *coordination media*, that is, the abstractions provided by coordination models for the coordination of multi-component systems.

So, the first point in the design of agent societies is the choice of the fittest coordination model—that is, the one providing the abstractions that are expressive enough to model the society interaction rules [46]. Thus, a society is designed around coordination media [47] embodying the interaction rules of its groups in terms of coordination rules (Figure 13.5). The behaviour of the suitably-designed coordination media, along with the behaviour of the agents playing social roles and interacting through such media, makes an agent society pursue its social tasks as a whole. This allows societies of agents to be designed as first-class entities.

The society model is described by means of the *Society Table* (Figure 13.7). Again, for each society, the required resources should also be specified.

13.2.3 The Environment Model

Resources are mapped onto *infrastructure classes*. So, an infrastructure class is first characterised by the services, the access modes, the permissions granted to roles and groups, and the interaction protocols associated to its resources.

The design of the components belonging to an infrastructure class may follow the most appropriate methodology for that class. Since **SODA** does not specifically address these issues, components like databases, expert systems, or security facilities, can all be developed according to the most suited specific methodology. Again, what is determined by **SODA** is the outcome of this phase, that is, the services to be provided by each infrastructure component, and its interfaces, as resulting from its associated interaction protocols.

Finally, **SODA** assumes that a topological model of the agent environment is provided by the designer—but does not provide for topological abstractions by itself, since any system and any application domain may call for different approaches to this problem.

The society model is described by means of the *Environment Table* (Figure 13.8)

Infrastructure Classes	Resource	Topological Abstraction	Policy
<i>infrastructure name</i>	<i>resource name</i>	<i>abstraction name</i>	<i>list of permissions</i>

Figure 13.8: Environment Table

13.2.4 Design: the outcome

In all, the results of the **SODA** design phase are expressed in terms of agent classes, societies of agents, and infrastructure classes. To summarise,

- an agent class is defined in terms of its individual and social roles and its interaction protocols, as well as the resources accessed and the relative access permissions
- a society of agents is defined in terms of its groups, as well as its corresponding coordination media and relative coordination rules, and resources
- an infrastructure class is defined in terms of its resources, as well as its topological abstraction and policies.

13.3 Meta-models

SODA is a methodology which explicitly focuses on suitably modelling *inter-agent* issues. As such, it assumes interaction to be the key aspect of its modelling process: a system entity appears in a **SODA** model only in that it is involved in some interactions.

Interaction is a major source of complexity in software systems. This is particular true in multi-agent systems, where interaction can take different forms: for instance, *social interaction* is concerned with agents interacting with each other, while *environmental interaction* regards the agents' interaction with their environment. So the purposes here are both to present the **SODA** meta-model – a deeper analysis of the meta-model is reported in Section 13.4 – and to exploit an agent-oriented methodology as a reference for stressing the pros and cons of different meta-modelling languages (Chapter 5). A methodology addressing only intra-agent issues would not fit: we need a methodology that deals with *inter-agent* issues, so that the social aspects of multi-agent systems are in the front line [123].

Therefore, in the following, first we define the **SODA** meta-model in UML (Subsection 13.3.1) and in OPM (Subsection 13.3.2), then we discuss the pros and cons of such meta-models and, by doing so, of the two approaches in general (Subsection 13.3.3).

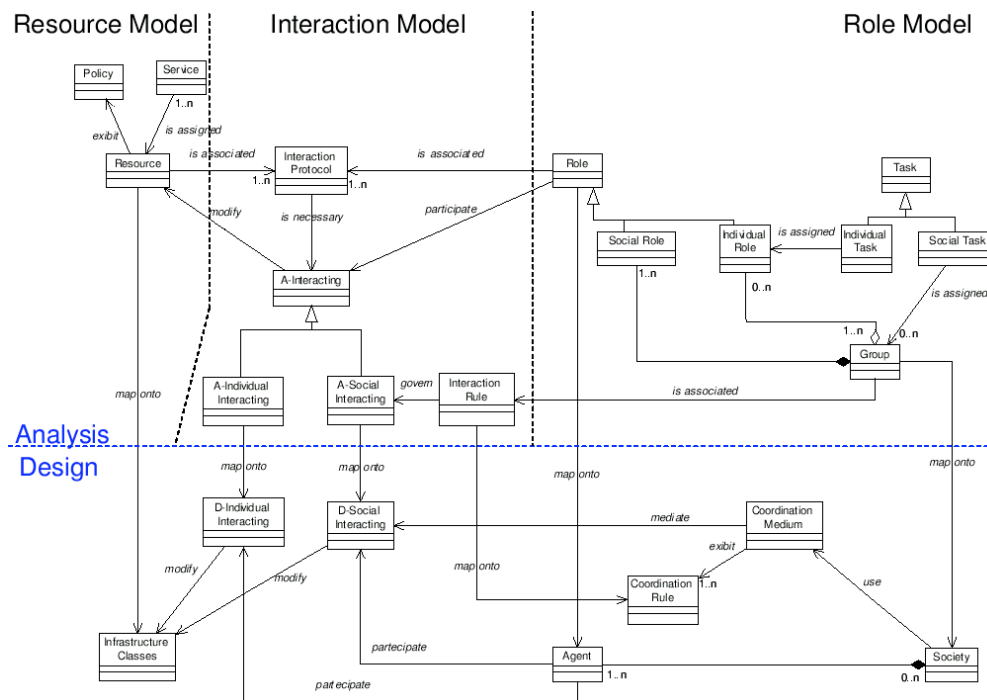


Figure 13.9: SODA Meta-model

13.3.1 Meta-model in UML

The meta-model of **SODA** (Figure 13.9) reflects the **SODA** distinction between the analysis phase (top) and the design phase (bottom) [123]. In the analysis phase, the *boundaries* between the models are well defined; in the design phase, instead, no such boundaries are shown, because the entities of the analysis sub-models do not map one-to-one onto analogous entities of the design model. It is worth noting that this meta-model clearly emphasises the *centrality of interaction* which is typical of **SODA**: in fact, if the interaction model were deleted, along with the corresponding classes in the design phase, concepts such as roles and resources would turn out to be separate and unrelated from one another.

Although this meta-model captures the **SODA** concepts and associations as far as UML's [143] graphical vocabulary makes it possible, the result is not completely satisfactory, for several reasons [123]. First, UML provides basically a unique type of concept/symbol (the class) to represent entities which are conceptually distinct in the meta-model. More precisely, while using the UML class notion to capture the **SODA organisational structure** – i.e., entities such as roles, tasks, groups, society, agents, resources, infrastructure classes – leads to a satisfactory representation of these aspects, the same cannot be said for *interaction*, whose classes are qualitatively different from the others (both in the analysis and in the design phase), as they try to model an intrinsically dynamic dimension by means of an intrinsically static abstraction.

The model entities are connected to each other by different relations—inheritance, composition, aggregation, and generic association. In particular, the relations between Group and (respectively) Individual role / Social role emphasise that a Social role may either coincide with an already defined Individual role (aggregation), or be defined *ex-novo* (composition). Moreover, the relations between the structural entities and the “interaction entities” are critical from the modelling viewpoint, since such entities are qualitatively different; this is why they are expressed by a generic (tagged) association.

Another key aspect concerns the connections from the analysis phase to the design phase. The label “map onto” is somehow vague, yet underlines the intrinsic difficulty in expressing the complex mapping from the analysis to the design phase via a single association link. For instance, when mapping Role onto Agent, the association itself is unable to express that Agent *inherits* task, permissions and interaction protocols from Role: so, a suitable label is the only (yet unsatisfactory) way to express this fact.

13.3.2 Meta-model in OPM

Figure 13.10 shows the **SODA** meta-model in OPM. Of course, many aspects discussed above – the distinction between the two phases, the analysis sub-models, the centrality of interaction, the association “map onto” – still hold: so, the overall model structure is basically the same as in Figure 13.9.

However, the richer expressiveness of OPM’s graphical vocabulary with respect to UML makes it possible to model the key aspect of interaction as an OPM *process*, rather than as a class, thus expressing the dynamic aspects that the (static) class notion alone could not capture. By doing so, the OPM meta-model of **SODA** captures the transient nature of interaction in much a better way than its UML counterpart. Furthermore, the richness of the OPM graphical vocabulary offers a better alternative to replace UML generic (tagged) associations with a new, semantically-clear symbology. For instance, the relation between Resource and Policy (and between Coordination Medium and Coordination Rule) now adopts a specific symbol to express that Policy not only has a structural relation with Resource, but is also an attribute of Resource.

On the other hand, since OPM introduces just one symbol (the solid black triangle) to represent both composition and aggregation, distinguishing between different relations (e.g Group/Individual Role, Group/Social Role) now requires a careful reading of the *participation constraint* of the relation (where * means “optional”, *m* means “many”, etc.).

Despite the richness of OPM’s vocabulary, some meta-modelling relations are still difficult to express: this is particularly true for the relations between structural entities and “X-Interacting” processes, that even the (several) object/process link types provided by OPM are unable to capture at a semantically-satisfactory level (more details in Section 13.3.3).

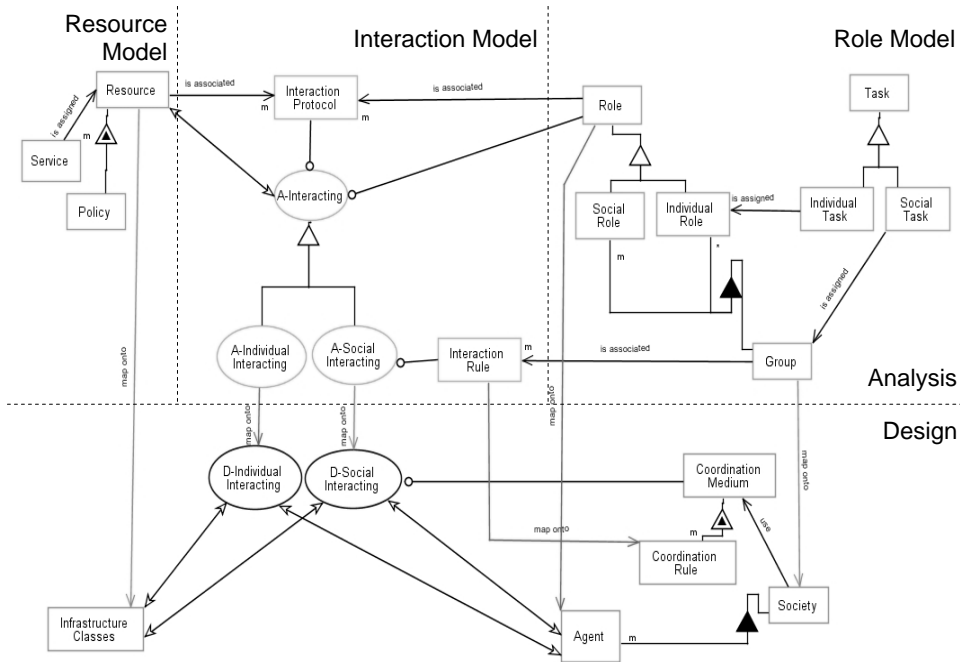


Figure 13.10: SODA Metamodel in OPM

13.3.3 Discussion

This section discusses and compares the **SODA** meta-models in UML and OPM, outlining the respective pros and cons. Generally speaking, both meta-models fall short in modelling the **SODA** concept of interaction and the relations between the structural parts and dynamic parts; in particular, this applies to the relation of “participation”, as outlined below.

Pros and Cons of SODA Meta-model in UML

The structural parts of the **SODA** methodology are well modelled. Due to its graphical vocabulary, UML is forced to model the **SODA** concept of interaction via its class notion, thus giving a static view of interaction, as if it were always present in the system—which is obviously misleading, since interaction has intrinsically a transient nature; indeed, capturing the transient aspects through a class diagram can be difficult.

On the other hand, UML enables the concept of “participation” in interaction to be expressed better than in OPM, thanks to the a generic tagged association: interestingly, this is possible just because interaction is represented as a class. However, distinguishing the semantic peculiarities of such associations based just on the label is not easy. For instance, although the same generic association for modelling the participation is used both in the analysis and in the design phases, in the first case the semantics is that Role

participates in Interaction, while in the latter this mean that not only Agent plays an active part in interaction, but its internal state is changed by interaction, too.

Pros and Cons of SODA Meta-model in OPM

As mentioned above, the main advantage of OPM with respect to UML concerns interaction modelling, which exploits OPM’s notion of process to represent the dynamic aspects. During the construction of the meta-model, however, the lack of a sort of “*tagged instrument link*” to connect objects and processes is perceived: currently, OPM’s instrument link is only untagged. Such a link would have been appreciable, for instance, to express that Role *participates* in the A-Interacting process—not just that it is necessary, as expressed by the standard instrument link. In fact, necessity is a static concept, while playing an active part in interaction, as Role does, implies dynamics. A similar problem emerges in the relation between the A-Social Interacting process and the Interaction Rule object, where it is not possible to express that Interaction Rule *governs* the social interaction—again, a more specific concept than just “being necessary”.

Analogously, in the design phase, an *Effect link* is used to represent the relations between the Agent object and the X-Interacting processes; this is semantically correct because the internal state of Agent is modified by interaction, but does not express the crucial fact that Agent takes an active part in interaction, while the Effect link just expresses that its internal state is modified as a consequence of interaction. As a last issue, in the relation between the D-Social Interacting process and the Coordination Medium object, we cannot express that the Coordination Medium *mediates* the social interaction by enacting the Coordination Rule—which, again, is more than just a mere “necessity”.

Summing up

Both UML and OPM are expressive enough to capture in their meta-model the structural parts of the SODA methodology: so, for instance, the role model and the resource model are expressed in a clear way, with a specific semantics. At the same time, as partially mentioned above, both approaches present some problems, the main one being that they fall short when asked to appropriately model the concept of interaction. In particular, the relation of participation, even though existing in both approaches, seems unable to capture the general concept of “participating in interaction” in a satisfactory way. This seems to indicate that while both UML and OPM methodologies are suitable to model the dynamic behaviour of *systems*, this ability is not conserved if they are used to build *meta-models* – actually, quite a different usage – although OPM is expressiveness under this viewpoint is a little better than UML’s.

So, the feeling is that neither OPM nor UML are fully adequate to capture the real essence of MAS methodologies, where interaction, in all its nuances – from a simple message exchange to mediated interaction via coordination media – is a key issue. In fact, suitably meta-modelling MAS methodologies seems to call for a specific approach,

which is able to model both the structural and the dynamic parts of a methodology, and to explicitly express the idea of participating in interaction.

13.4 Limitations

Meta-models, as explained in Chapter 6, are useful tools because they enable checking and verifying the completeness and expressiveness of a methodology by understanding its deep semantics. In the case of **SODA**, the analysis of its meta-models highlights different limitations in the methodology.

First of all, the environment is not designed in a good way even if **SODA** could be enrolled in the strong-env category (Chapter 8). In fact **SODA** deals with the environment since the analysis phase and it provides support both for the modelling and design of the environment. However the environment abstraction adopted by **SODA** – infrastructure classes – are too simple to express the complexity of the environment of MAS. The role of the environment in cognitive agency is mainly concerned with the agent’s cognitive model of the environment, the agent’s action over the environment, and the practical reasoning over these actions [220]. According to Weyns and Omicini [220] the role of the environment in cognitive agent systems could be considered from five different perspectives: (*i*) the environment as a container and a means for communication, (*ii*) the environment as an organisational layer, (*iii*) the environment as a coordination infrastructure for cognitive agents, (*iv*) Markovian environments, and finally (*v*) task environments. Infrastructure classes address only (*i*), in fact they represent the world of the resources of the MAS and are related to the topology abstractions—that are not designed yet. The third perspective, instead, is addressed by **SODA** by means of the coordination media designed in the society model. These media are thought of as a piece of the organisational structure of the system rather than as a part of the environment. This leads also to a wrong representation of the environment: the environment is conceptually designed in two different and “unrelated” times. In addition, two different environment abstractions (Section 2.3) are adopted for representing the environment and each one is not expressive enough for capturing the whole environment.

Second, the interactions are not expressed best since they are strictly tied to protocols, which are not suitable in the context of cognitive agents. A cognitive agent is typically presented as able to perform actions over the environment, where the environment represents everything outside the agent [220]. A good interaction model should be able to express the actions that agents can perform, not just the protocols underpinning them. In the same way, interaction rules are not expressive enough in order to shape and bound the agent’s interaction space. In fact, these rules support coordination in agent societies, but they do not allow the expression of constraints over the actions that agents are able to do. These constraints are necessary in open MASs where self-interested agents could try to violate system security.

Another limitation of **SODA** is that the methodology captures only a part of the whole software development process. The methodology, in fact, addresses only the analysis and design phases, while the requirements capturing and analysis, and the architectural design require a different approach. The adoption of a different approach for the requirements capturing and analysis is not so difficult because this applies before the beginning of the **SODA** process; for example the traditional use case technique [110] has been fruitfully adopted. Indeed, the adoption of a different approach for the architectural design is not straightforward, since it should “break” the **SODA** process and it should adopt the same abstractions exploited by **SODA**.

Finally, the engineering of non-trivial MASs requires principles and mechanisms for a multi-layered description, which could be used by MAS designers to provide different levels of abstraction over MASs as illustrated in Chapter 12. **SODA** in its first formulation does not provide any support for the scalability of the system representation.

14

SODA: The New Version

This chapter presents the new version of **SODA** that represents the work of my Ph.D. The chapter is organised as follows: Section 14.1 presents the motivations that have led to the reformulation of the methodology. Then the **SODA** meta-model and the layering principle adopted by **SODA** are depicted respectively in Section 14.2 and Section 14.3, the analysis phase and the design phase are illustrated in Section 14.4 and Section 14.5. Finally Section 14.6 reports a summary of the chapter.

14.1 Motivations

SODA [144, 127, 3, 128] has recently been extended to address the limitations of the previous version (Section 13.4). The reformulation process begins from the meta-model presented in the Chapter 13 which has been redrawn from scratch. Subsequently the features of the chosen abstractions are described by means of a relational table.

In particular the A&A meta-model (Chapter 7) is adopted in order to improve the environment modelling and design, and a new mechanism to manage the complexity of system description is proposed (Section 14.3). The interactions, that are again the core of the methodology, have been fully redesigned. In addition, **SODA** is reformulated by organising it in two phases, each structured in two sub-phases: the *Analysis phase*, which includes the Requirements Analysis and the Analysis steps, and the *Design phase*, including the Architectural Design and the Detailed Design steps. In this way the requirements analysis and the architectural design are now part of the **SODA** process and they do not require the ad hoc approach as in the first **SODA** definition. An overview of the methodology structure is shown in Figure 14.1: each step is practically described in terms of a set of relational tables, listed in the figure.

SODA provides only a tabular representation of the system, as in the first formulation, however the tables have been fully redesigned. Each table in the methodology has both a full name and a unique acronym, which specifies the layer where the table belongs (in round brackets) and the involved entities. For instance, $(L)AR_t$ refers to the Actor (specified by the “A”)-Requirement (specified by the “R”) table (specified by the “t”) at layer L.

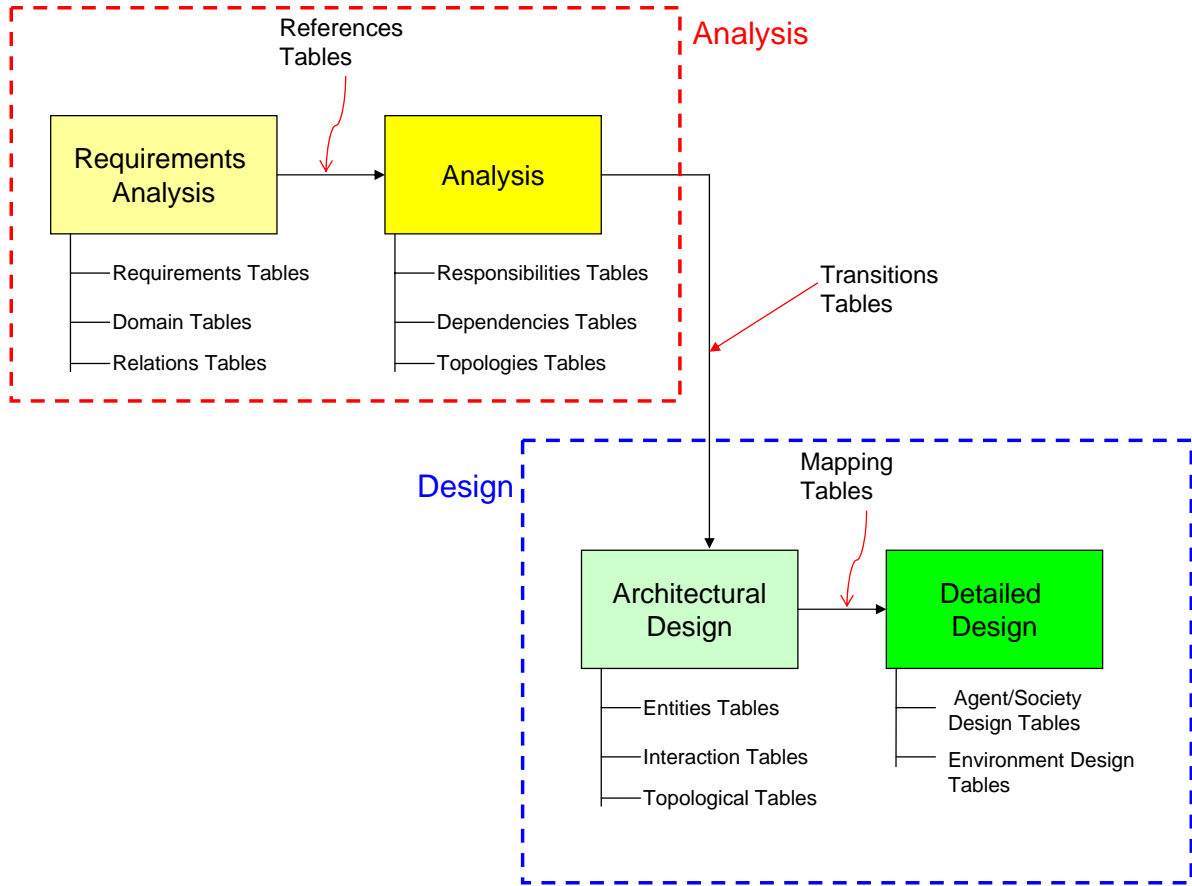


Figure 14.1: An overview of the SODA process

14.2 The New Meta-model

The meta-model that represents the abstract entities adopted by SODA is depicted in Figure 14.2.

Requirements Analysis. Several abstract entities are introduced for requirement modelling (see Figure 14.2 “requirements analysis” part): in particular, *requirement* and *actor* are used for modelling the customers’ requirements and the requirement sources, respectively, while the *external-environment* notion is used as a container of the *legacy-systems* that represent the legacy resources of the environment. The relationships between requirements and legacy systems are then modelled in terms of suitable *relation* entities.

Analysis. The Analysis step expresses the abstract requirement representation in terms of more concrete entities such as *tasks* and *functions* (see Figure 14.2, “analysis” part). Tasks are activities requiring one or more competences, while functions are reactive activities aimed at supporting tasks. The relations highlighted in the previous step are

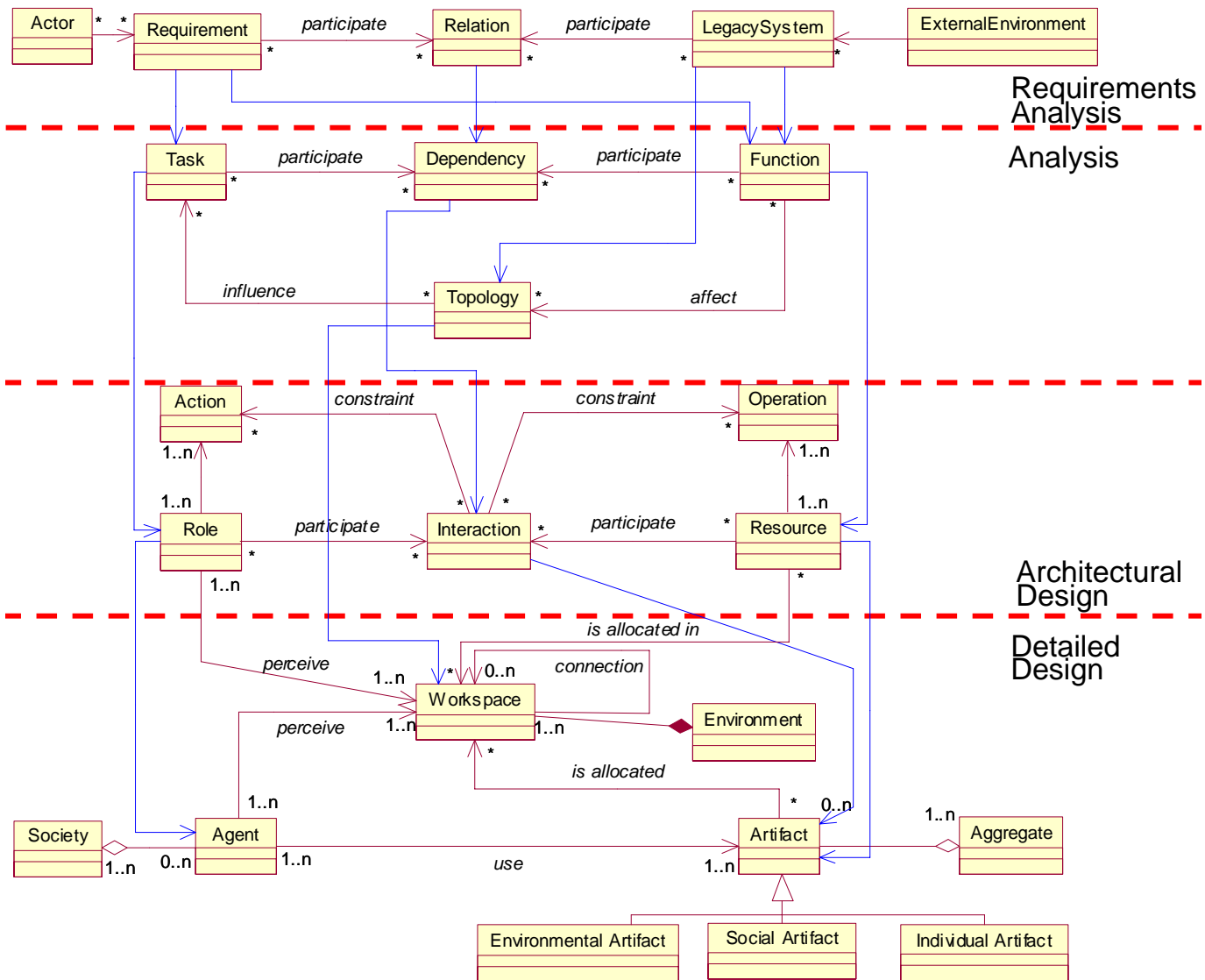


Figure 14.2: SODA Meta-model

now the starting point for the definition of *dependencies* (interactions, constraints, etc.) among the abstract entities. The structure of the environment is also modelled in terms of *topologies*, i.e. topological constraints over the environment.

Topologies are often derived from functions, but can also constrain / affect task achievement.

Architectural Design. The main goal of this stage is to assign responsibilities of achieving tasks to *roles*, and responsibilities of providing functions to *resources* (see Figure 14.2, “architectural design” part). To this end, roles should be able to perform *actions*, and resources should be able to execute *operations* providing one or more functions. The dependencies identified in the previous phase become here *interactions*, i.e. “rules” enabling and bounding the entities’ behaviour. Finally, the topology constraints lead to the definition of *workspaces*, i.e. conceptual places structuring the environment.

Detailed Design. Detailed Design is expressed in terms of *agents*, agent *societies*, *artifacts* and *aggregates* (see Figure 14.2 “detailed design” part). Agents are intended here as autonomous entities able to play several roles, and the resources identified in the previous step are now mapped onto suitable artifacts. In the meta-model the artifact is reported specifying its “type” – individual, social and environmental – derived from the taxonomy presented in Subsection 7.2.2. it in an easier way. We recall here the taxonomy already shown:

- Individual artifact handles the interaction of a single agent within a MAS, and essentially works as a mediator between the agent and the MAS itself. Since they can be used to shape of admissible interactions of individual agents in MAS, individual artifacts play an essential role in engineering both organisational and security concerns in MAS.
- Environmental artifact brings an external resource within a MAS, by mediating agent actions and perceptions over resources. As such, environmental artifacts play an essential role in enabling, disciplining and governing the interaction between agents and MAS environment.
- Social artifact rules social interactions within a MAS—even though indirectly, since it technically mediates interactions between individual, environmental, and possibly other social artifacts. Social artifacts in **SODA** play the role of the coordination artifacts that embody the rules around which societies of agents can be built.

In **SODA** a society can be seen as a group of interacting agents and artifacts when its overall behaviour is essentially an autonomous, proactive one; it can be seen an aggregate when its overall behaviour is essentially a functional, reactive one.

The *workspaces* defined in the Architectural Design step take now the form of an open set of artifacts and agents—that is, artifacts can be dynamically added to or removed from workspaces, and agents can dynamically enter (join) or exit workspaces.

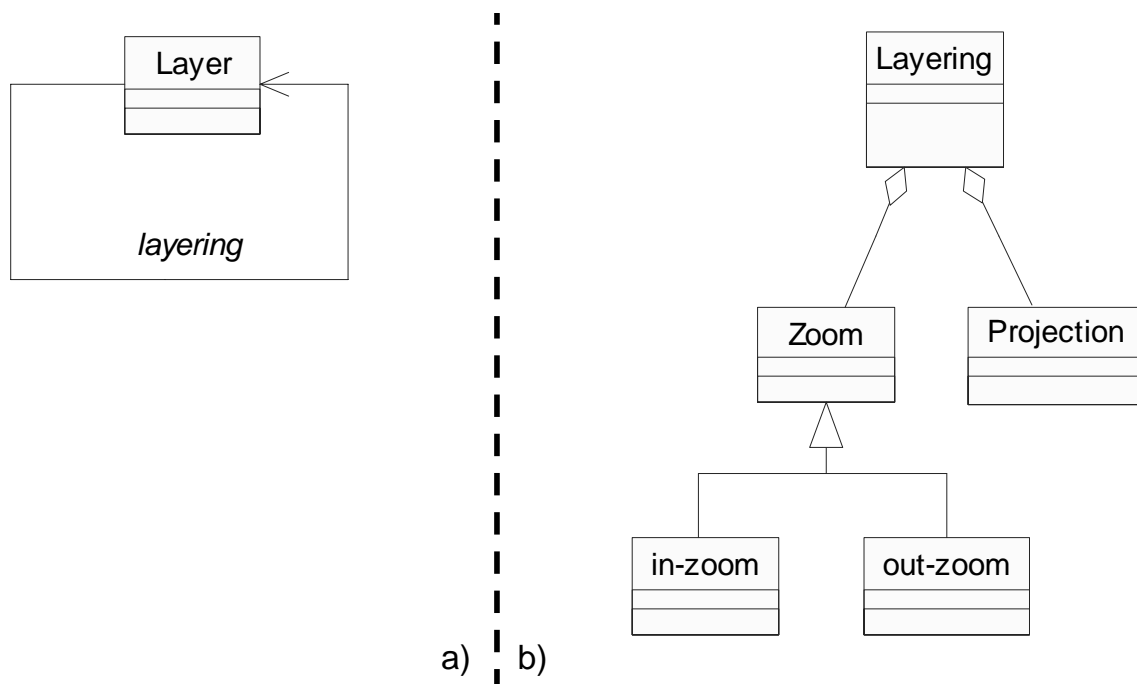


Figure 14.3: Meta-models of the layering principle (left) and of the layering mechanisms (right)

14.3 Layering

Following the principles sketched in Chapter 12, a simple layering principle is introduced in **SODA**: the system is represented as composed by different layers of abstraction and it is possible to move from one layer to another layer by means of a *layering operation* (Figure 14.3 part b)).

This layering operation consists of two mechanisms (Figure 14.3 part b)), *zoom* and *projection*: *zoom* makes it possible to pass from an abstract layer to another, while *projection* projects the entities of a layer into another. *Zoom* is the only mechanism relating layers to each other: more precisely, in-zooming the entities of a (more abstract) layer leads to a more detailed layer, while out-zooming the entities of a (more detailed) layer leads back to to a more abstract layer.

Of course, not all the entities of a layer should be in-zoomed in another (more detailed) layer at the same time, since this could easily lead to mistakes; so, in order to preserve the internal consistency of each layer, the entities of the more abstract layer that are not in-zoomed are projected (i.e, made available) in the more detailed layer “as they are” by the projection mechanism (see Figure 14.4).

In general, when working with **SODA**, the starting layer, called *core layer*, is labelled with “C” and is always *complete*—that is, it contains all the entities required to fully

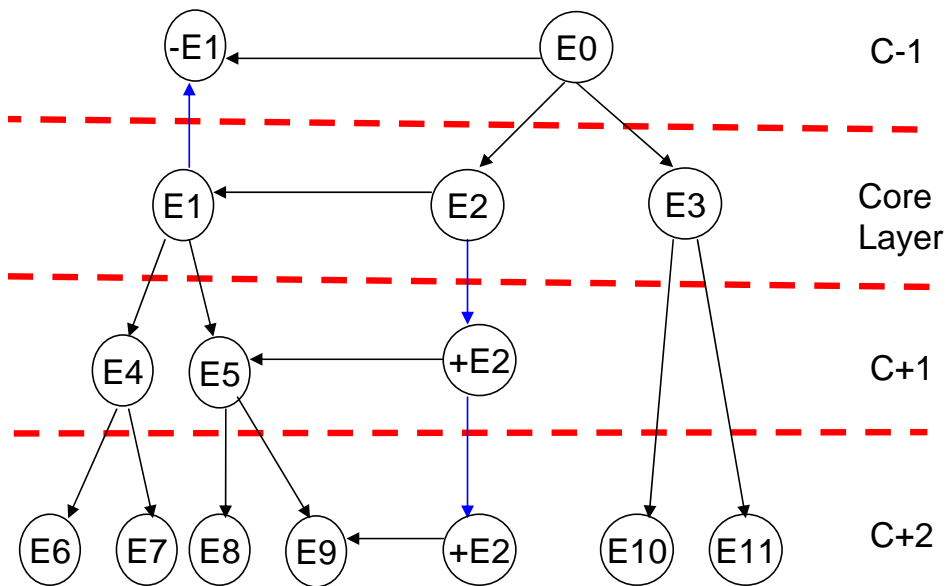


Figure 14.4: Layers in SODA

describe a given abstract layer. Any other layer contains just *i*) the entities that have been possibly (in/out-) zoomed from another layer, as well as *ii*) the entities possibly projected “as they are” from other layers: so, in general, these layers are not necessarily complete—though of course they might be so, as in the case of layer C+2 in Figure 14.4.

The layering principle is represented by means of a Zooming Table $((C)Z_t)$ (Figure 17.2) The Zooming Table formalises the in-zoom of a layer into the more detailed layer; of course, the same table can be used to represent the dual out-zoom process.

Layer L	Layer L+1
<i>out-zoomed entity</i>	<i>in-zoomed entities</i>

Figure 14.5: $(L)Z_t$

14.4 The Analysis Phase

This section presents the Analysis phase of SODA. In particular Subsection 14.4.1 explains the tables of the Requirements Analysis, Subsection 14.4.2 shows the tables for moving from Requirements Analysis to Analysis, and Subsection 14.4.3 presents the tables of the Analysis step.

14.4.1 Requirements Analysis

The goal of Requirements Analysis is the characterisation of both the customers' requirements and the legacy systems with which the system should interact, as well as to highlight the relationships among requirements and legacy systems.

The Requirements Analysis step consists of three sets of tables: Requirements Tables, Domain Tables and Relations Tables.

Requirements Tables (Figure 14.6) define the abstract entities tied to the concept of "requirement": in particular, the Actor-Requirement table $((L)AR_t)$ specifies the collection of the requirements associated to each actor, while the Requirement table $((L)Re_t)$ describes each single requirement.

Actor	Requirement
<i>actor name</i>	<i>requirement names</i>
Requirement	Description
<i>requirement name</i>	<i>requirement description</i>

Figure 14.6: Requirements Tables, in top-down order: $((L)AR_t)$, $((L)Re_t)$

Domain Tables (Figure 14.7) define the abstract entities tied to the concept of "external environment". This group of tables is composed of the ExternalEnvironment-LegacySystem table $((L)EELS_t)$, which specifies the legacy systems associated to the external environment, and the LegacySystem table $((L)LS_t)$, which describes each single legacy system.

External-Environment	Legacy-System
<i>external-environment name</i>	<i>Legacy-System names</i>
Legacy-System	Description
<i>legacy-system name</i>	<i>legacy-system description</i>

Figure 14.7: Domain Tables, in top-down order: $((L)EELS_t)$, $((L)LS_t)$

Finally Relations Tables (Figure 14.8) link the abstract entities with each other. In particular, the Relation table $((L)Rel_t)$ describes all the relationships among abstract entities, while the Requirement-Relation table $((L)RR_t)$ specifies the relations where each requirement is involved, and the LegacySystem-Relation table $((L)LSR_t)$ specifies the relations where each legacy-system is involved.

Relation	Description
<i>relation name</i>	<i>relation description</i>
Requirement	Relation
<i>requirement name</i>	<i>relation names</i>
Legacy-System	Relation
<i>legacy-system name</i>	<i>relation names</i>

Figure 14.8: Relations Tables, in top- down order: $(L)Rel_t$, $(L)RR_t$, $(L)LSR_t$.

14.4.2 From Requirements Analysis to Analysis

In order to move from Requirements Analysis to Analysis, the relations between the different abstractions adopted in the two steps must be precisely identified: this is done by means of the References Tables (Figure 14.9).

Requirement	Task
<i>requirement name</i>	<i>task names</i>
Requirement	Function
<i>requirement name</i>	<i>function names</i>
Legacy-System	Function
<i>legacy-system name</i>	<i>function names</i>
Legacy-System	Topology
<i>legacy-system name</i>	<i>topology names</i>
Relation	Dependency
<i>relation name</i>	<i>dependency names</i>

Figure 14.9: References Tables, in top-down order: $(L)RRT_t$, $(L)RRF_t$, $(L)RLSF_t$, $(L)RLST_t$, $(L)RRD_t$.

In particular, *i*) the Reference Requirement-Task table $((L)RRT_t)$ specifies the mapping between each requirement and the generated tasks, the Reference Requirement-Function table *ii*) $((L)RRF_t)$ specifies the mapping between each requirement and the generated functions; *iii*) the Reference LegacySystem-Function table $((L)RLSF_t)$, which specifies the mapping between each legacy-system and the corresponding functions; *iv*) the Reference LegacySystem-Topology table $((L)RLST_t)$, which specifies the mapping between legacy-systems and topologies; and *v*) the Reference Relation-Dependency table $((L)RRD_t)$, which specifies the mapping between relations and dependencies.

14.4.3 Analysis

The Analysis step expresses the abstract requirement representation defined in the previous step in terms of more concrete entities such as *tasks* and *functions*. Functions can come both from the legacy-system entities or be designed ex-novo: in the former case they should just be modelled, in a sort of reverse engineering step, while in the latter they have to modelled (and later designed) as brand-new functionalities.

The Analysis step exploits three sets of tables: Responsibilities Tables, Dependencies Tables and Topologies Tables. Responsibilities Tables (Figure 14.10) define the abstract entities tied to the concept of “responsibilities centre”—namely, tasks and functions. So, this set of tables includes the Task table $((L)T_t)$, which lists all the tasks, and the Function table $((L)F_t)$, which lists all the functions.

Task	Description
<i>task name</i>	<i>task description</i>
Function	Description
<i>function name</i>	<i>function description</i>

Figure 14.10: Responsibilities Tables, in top-down order: $(L)T_t$, $(L)F_t$

Dependencies Tables (Figure 14.11) relate functions and tasks with each other. More precisely, the Dependency table $((L)D_t)$ describes all the dependencies among abstract entities, while the Task-Dependency table $((L)TD_t)$ specifies the set of dependencies where each task is involved, and the Function-Dependency table $((L)FD_t)$ specifies the list of dependencies where each function is involved. Typically, when a requirement generates both a task and a function, the function is necessary to achieve the task. Correspondingly, other dependencies arise, in addition to the dependencies originating from the **SODA** relations.

Dependency	Description
<i>dependency name</i>	<i>dependency description</i>
Task	Dependency
<i>task name</i>	<i>dependency names</i>
Function	Dependency
<i>function name</i>	<i>dependency names</i>

Figure 14.11: Dependencies Tables in top-down order $(L)D_t$, $(L)TD_t$ and $(L)FD_t$.

Topologies Tables (Figure 14.12), in turn, express the topological constraints over the environment. So, the Topology table $((L)Top_t)$ describes the topological constraints,

while the Task-Topology table $((L)TTop_t)$ specifies the list of the topological constraints which affect the task, and the Function-Topology table $((L)FTop_t)$ specifies the list of the topological constraints affected by the function.

Topology	Description
<i>Topology name</i>	<i>topology description</i>
Task	Topology
<i>task name</i>	<i>topology names</i>
Function	Topology
<i>function name</i>	<i>topology names</i>

Figure 14.12: Topologies Tables in top-down order: $(L)Top_t$, $(L)TTop_t$, $(L)FTop_t$.

14.5 The Design Phase

This section presents the Design phase of **SODA**. In particular Subsection 14.5.1 shows the tables for moving from Analysis to Architectural Design, Subsection 14.5.2 explains the tables of the Architectural Design, Subsection 14.5.3 shows both the carving operation and the tables for moving from Architectural Design to Detailed Design, and Subsection 14.5.4 presents the tables of the Detailed Design step.

14.5.1 From Analysis To Architectural Design

In order to link the Analysis step with the Architectural Design step, the Analysis entities are related to the Architectural Design by means of Transition Tables (Figure 14.13). So, for each layer, the Transition Role-Task table $((L)TRT_t)$ relates tasks and roles, the Transition Resource-Function table $((L)TRF_t)$ links functions and resources, the Transition Interaction-Dependency table $((L)TID_t)$ maps interactions onto dependencies, and the Transition Topology-Workspace table $((L)TTopW_t)$ specifies the mapping between topologies and workspaces.

14.5.2 Architectural Design

At this stage, the main goal is to assign responsibilities of achieving tasks to *roles*, and responsibilities of providing functions to *resources*. The Architectural Design step consists of three sets of tables: Entities Tables, Interaction Tables and Topological Tables.

Entities Tables (Figure 14.14) describe both the active entities (the roles) able to perform some action in the system, and the passive entities (the resources) which provide services. In particular, the Action table $((L)A_t)$ describes the actions executable by some

Role	Task
<i>role name</i>	<i>task names</i>
Resource	Function
<i>resource name</i>	<i>function names</i>
Dependency	Interaction
<i>dependency name</i>	<i>interaction names</i>
Topology	Workspace
<i>topology name</i>	<i>workspace names</i>

Figure 14.13: Transition Tables, in top-down order: $(L)TRT_t$, $(L)TRF_t$, $(L)TID_t$, $(L)TTopW_t$.

role, while the Operation table $((L)O_t)$ specifies the operations provided by resources. Then, the Role-Action table $((L)RA_t)$ specifies the actions that each role can do, while the Resource-Operation table $((L)RO_t)$ specifies the operations that each resource can provide.

Action	Description
<i>action name</i>	<i>description</i>
Operation	Description
<i>operation name</i>	<i>description</i>
Role	Action
<i>role name</i>	<i>action names</i>
Resource	Operation
<i>resource name</i>	<i>operation names</i>

Figure 14.14: Entities Tables, in top-down order: $(L)A_t$, $(L)O_t$, $(L)RA_t$, $(L)RO_t$

Interaction Tables (Figure 14.15) describe the interaction between roles and resources: more precisely, the Interaction table $((L)I_t)$ defines the single interactions, the Role-Interaction table $((L)RoI_t)$ specifies the interactions where each role is involved, and the Resource-Interaction table $((L)ReI_t)$ specifies the interactions where each resource is involved.

Finally, Topological Tables (Figure 14.16) describe the logical structure of the environment. More precisely, the Workspace table $((L)W_t)$ describes the workspaces, the Workspace-Connection table $((L)WC_t)$ shows the connections among the workspaces of a given layer (the hierarchical relations between workspaces are expressed via the Zoom-

Interaction	Description
<i>interaction name</i>	<i>description</i>
Role	Interaction
<i>role name</i>	<i>interaction names</i>
Resource	Interaction
<i>resource name</i>	<i>interaction names</i>

Figure 14.15: Interaction Tables, in top- down order: $(L)I_t$, $(L)RoI_t$, $(L)ReI_t$

ing Table), and the Workspace-Resource table ($(L)WRe_t$) shows the allocation of the resources to workspaces. It should be noted that a resource could be allocated in several different workspaces at the same time. For instance, a single, distributed resource can in principle be used to model a distributed service, accessible from multiple nodes of a network. Similarly, the Workspace-Role table ($(L)WRo_t$) lists the workspaces that each role can perceive in the system.

Workspace	Description
<i>workspace name</i>	<i>description</i>
Workspace	Connection
<i>workspace name</i>	<i>workspace names</i>
Workspace	Resource
<i>workspace name</i>	<i>resource names</i>
Role	Workspace
<i>role name</i>	<i>workspace names</i>

Figure 14.16: Topological Tables, in top-down order: $(L)W_t$, $(L)WC_t$, $(L)WRe_t$ and $(L)WRo_t$

14.5.3 From Architectural Design to Detailed Design

The goal of Detailed Design is to choose the most adequate representation level for each architectural entity, thus leading to depict one (detailed) design from the several potential alternatives outlined above. For the sake of concreteness, let us refer to Figure 14.17 (left), where the hypothesis that the Architectural Design phase outlined roles R1, R2 at the core layer C, roles R4, R5 and the projection of R2 at layer C+1, and roles R6, R7, R8 and R9 at layer C+2, is made. Turning this conceptual view into a real design view means to choose one representation (i.e., zoom) level for each entity, starting from the

core layer: so, for instance, we could decide to zoom only R1, keeping R2 at the basic (core) representation level. Moreover, we could choose to further in-zoom R4 as the set of (sub)roles R6,R7, while keeping R5 as is. The result can be graphically expressed by *carving out* the roles R1, R2, R4, R5, R6 and R7 from the Architectural Design view, as shown with the curbed line in Figure 14.17 (centre).

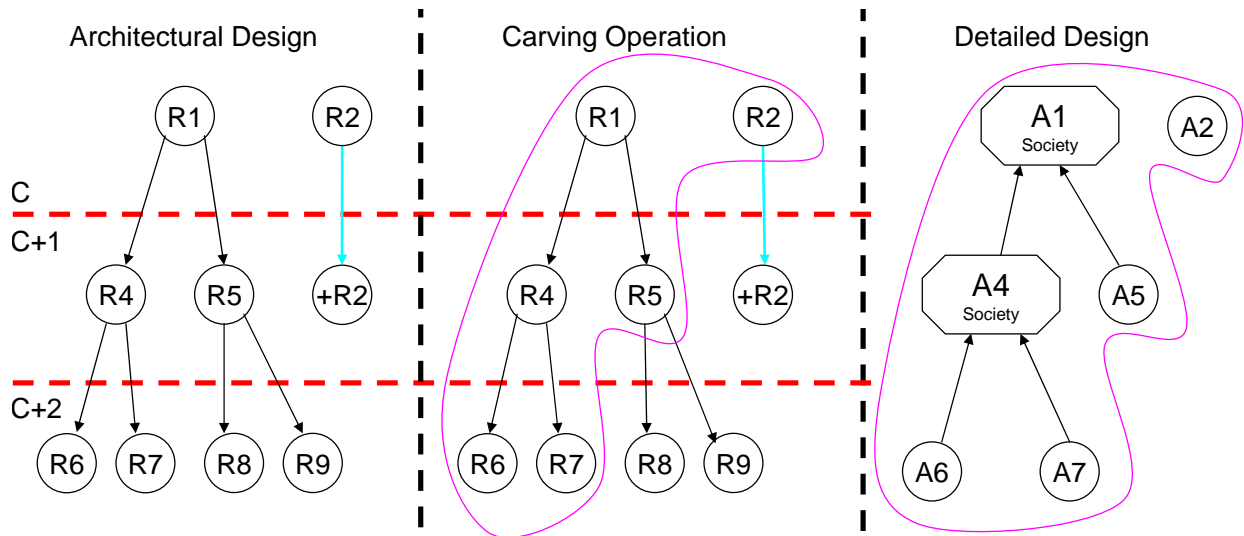


Figure 14.17: Design steps and Carving Operation

A similar approach is adopted for the environmental entities: the un-zoomed resources identified in the previous step are now mapped onto suitable artifacts (intended as entities providing some services), and in-zoomed resources are mapped onto aggregates of artifacts.

This “carving operation” represents the boundary between Architectural Design – expressed in terms of roles, services, resources and workspaces – and Detailed Design—expressed in terms of agents, agent societies, artifacts and aggregates. In the case of our example (see Figure 14.17 (right)), role R1 is to be mapped onto an agent society (A1), while role R2 is to be mapped onto an individual agent (A2). Going further, the agent society A1 is composed of two entities, representing roles R4 and R5: again, the first is to be mapped onto an agent society (A4), since role R4 is in-zoomed in the carving, while R5 is to be mapped onto an individual agent (A5); the same process applies to A4, which turns out to be composed of agents A6 and A7, mapping roles R6 and R7, respectively.

The result is the set of Mapping Tables (Figure 14.18), which relate the Architectural Design entities to the Detailed Design. In particular, the Mapping Agent-Role table ($((L)MAR_t)$) maps roles onto agents, the Mapping Artifact-Resource table ($((L)MArR_t)$) maps resources onto artifacts, and the Mapping Artifact-Interaction table ($((L)MArI_t)$) maps the rules specified in the Architectural Design onto the artifacts that implement and enforce them.

In order to support and simplify the designer work, in the tables the “kind” of the artifact – as reported in Subsection 7.2.2 – is specified. For example in table $(L)MAR_t$ (Figure 14.18) the resources are associated to environmental artifact, in table $(L)MArR_t$ (Figure 14.18) the interactions are mapped to social artifact and the same social artifact is associate to society in table $(L)SAr_t$ (Figure 14.19). Finally in table $(L)AA_t$ (Figure 14.19) agents are associated to individual artifact.

Agent	Role
<i>agent name</i>	<i>role names</i>
(Environmental) Artifact	Resource
<i>artifact name</i>	<i>resource names</i>
Interaction	(Social) Artifact
<i>interaction name</i>	<i>artifact names</i>

Figure 14.18: Mapping Tables in top- down order: $(L)MAR_t$, $(L)MArR_t$, $(L)MArI_t$

14.5.4 Detailed Design

The Detailed Design step exploits two sets of tables: Agent/Society Design Tables, and Environment Design Tables. The first set of tables (Figure 14.19) depicts agents, individual artifacts, and the agent societies derived from the carving operation. More precisely, the Agent-Artifact table ($(L)AA_t$) specifies the individual artifacts related to each agent, the Society-Agent table ($(L)SA_t$) lists the agents belonging to a specific society, and the Society-Artifact table ($(L)SAr_t$) specifies the social artifacts related to each agent society.

Agent	(Individual) Artifact
<i>agent name</i>	<i>artifact names</i>
Society	Agent
<i>Society name</i>	<i>agent names</i>
Society	(Social) Artifact
<i>society name</i>	<i>artifact names</i>

Figure 14.19: Agent/Society Design Tables in top- down order: $(L)AA_t$, $(L)SA_t$, $(L)SAr_t$

In turn, Environment Design Tables concern the design of artifacts and workspaces: the Artifact-UsageInterface table ($(L)AUI_t$) details the operations provided by each artifact, the Aggregate-Artifact table ($(L)AggA_t$) specifies which artifacts are part of the Aggregate generated by the carving, while the Workspace-Artifact table ($(L)WA_t$) specifies the location of artifacts in the workspace. Here the distinction between the artifact

types is not presented because the design of the usage interface, the allocation of the artifacts to workspaces and the aggregates are done in the same way for all the artifact's types.

Artifact	Usage Interface
<i>artifact name</i>	<i>list of operations</i>
Aggregate	Artifact
<i>aggregate name</i>	<i>artifact names</i>
Workspace	Artifact
<i>Workspace name</i>	<i>artifact names</i>

Figure 14.20: Environment Design Tables in top- down order: $(L)AUI_t$, $(L)AggA_t$, $(L)WA_t$

14.6 Summing up

This chapter has introduced the new version of the **SODA** methodology. The limitations of the previous version of **SODA** are now resolved by means of the introduction of the A&A meta-model and of the layering principle.

As a conclusion, here we presents an evaluation of **SODA** as regards the criteria proposed in Section 4.3:

- Lifecycle criteria
 - *Development Lifecycle*: iterative.
 - *Coverage Lifecycle*: analysis and design.
 - *Development Perspective*: middle-out (Section 12.1).
 - *Support for verification*: no.
- Notation criteria
 - *Notation*: ad hoc.
 - *Easy to understand*: high.
 - *Usability*: medium.
 - *Supporting tool*: no.

15

The SODA Process

This chapter presents the **SODA** process modelled by SPEM version 1.1 (Subsection 5.2.2). The “silver bullet”, one-size-fits-all methodology is nowadays a recognised *chimera* in the Software Engineering field. Quite naturally, each SE methodology has its own approach and peculiarities, its pros and cons—and none is general enough to fit with any possible application scenario (Chapter 4). As a consequence, it has been observed that software designers tend to define/refine their own problem-specific methodology, as an *ad hoc* tool specially suited to the typical application scenarios of interest. Of course, such methodological approaches can be hardly reused in different contexts, or for different problems, without significant changes: in the overall, this results in considerable costs of the system development process.

For these reasons, research efforts are ongoing meant to define a unified meta-model, aimed at representing the existing methodologies in a uniform way, so as to promote their mutual comparison, their composition and reuse—this area is sometimes referred to as *Method Engineering* [15]. SPEM and OPEN (Subsection 5.2.2) are two key references for this purpose: as it could be expected, both were conceived for an object-oriented context, since most current methodologies adopt this paradigm as the reference one. In particular, SPEM seems a natural candidate for representing the meta-models of Software Engineering methodologies, both because it is an OMG standard, and because it is based on formal descriptions that can lead to consistent, comparable models: so, an interesting challenge is to test its applicability to other, non object-oriented Software Engineering domains.

In this chapter, in particular, we explore its applicability to the **SODA** methodology, whose abstractions and mechanisms are particularly suited to the design and development of complex software systems. While some AO methodologies have already been modelled in SPEM by the FIPA Methodology Technical Committee (Chapter 6), here we mean to exploit SPEM to model the **SODA** methodology process, taken as a significant case study for stressing SPEM’s strengths and weaknesses because of its specific features—namely, its focus on modelling the social issues and the application environment, and its mechanisms for capturing the layered structure of complex systems. In particular, Section 15.1 presents the general **SODA** process, Section 15.2 presents the detail of the Analysis phase and

Section 15.3 shows the Design phase. Section 15.4 presents an evaluation of the adoption of SPEM for modelling **SODA**.

15.1 The process

The **SODA** process is composed of two disciplines (see Figure 15.1):

- *Analysis*: it includes the requirements analysis and the system analysis;
- *Design*: it includes the architectural design and the detailed design.

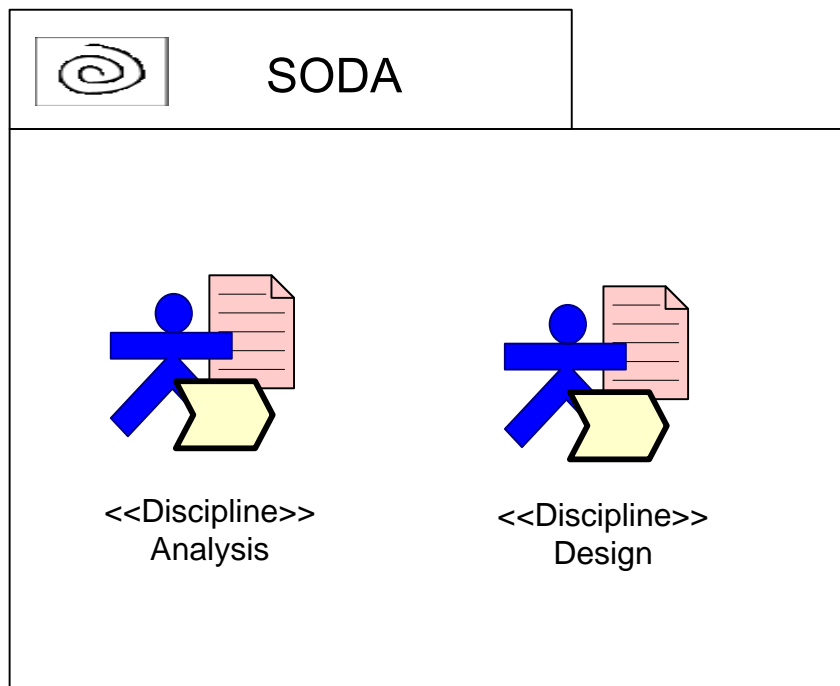


Figure 15.1: The disciplines of the **SODA** process

Figure 15.2 shows the complete **SODA** process. Each phase of **SODA** produces a system model (Analysis Model and Design Model) constituted by a set of relational tables. **SODA** is an iterative process, so when each phase is terminated it is possible to re-start the phase or also re-start the previous phase.

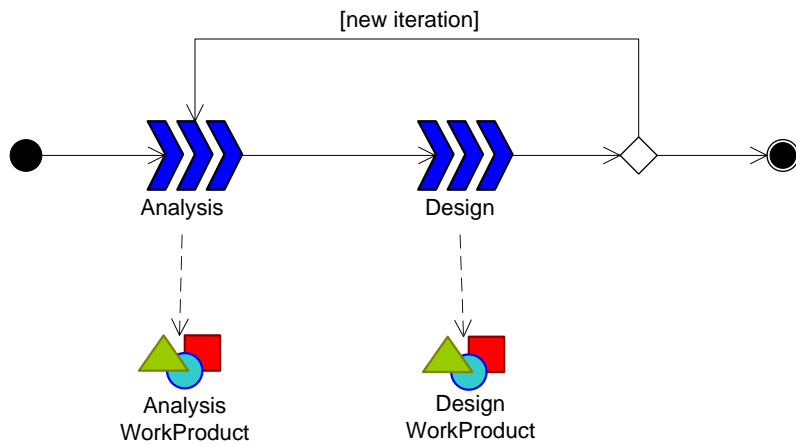


Figure 15.2: The complete **SODA** process

Each phase of the **SODA** process is composed by two steps as showed in Figure 15.3. Since the Phase element of SPEM could be decomposed in a set of WorkDefinition [200], it is possible to model each step of the **SODA** process by means of WorkDefinition elements and the *include* relation between Phase and WorkDefinition elements.

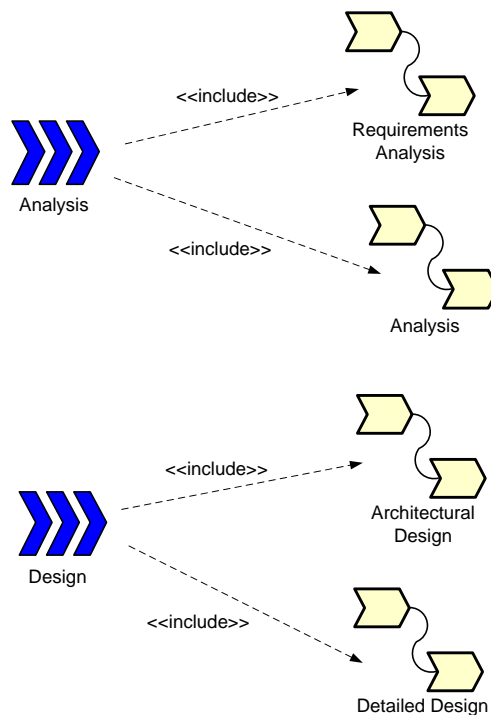


Figure 15.3: Use-case diagram of the **SODA** process

15.2 The Analysis Phase

This section presents the Analysis phase process. In particular Subsection 15.2.1 presents the Analysis discipline, Subsection 15.2.2 details the Requirements Analysis step, while Subsection 15.2.3 details the Analysis step. Finally Subsection 15.2.4 details the Analysis Model.

15.2.1 The Analysis Discipline

The *Analysis* discipline (Figure 15.4) can be naturally characterised by three actors (*Roles*): one is responsible for the activities of *Requirements Analysis* step, another for the activities of the *Analysis* step, while the third is an application-domain expert aimed at assisting the two previous actors when analysing the application domain.

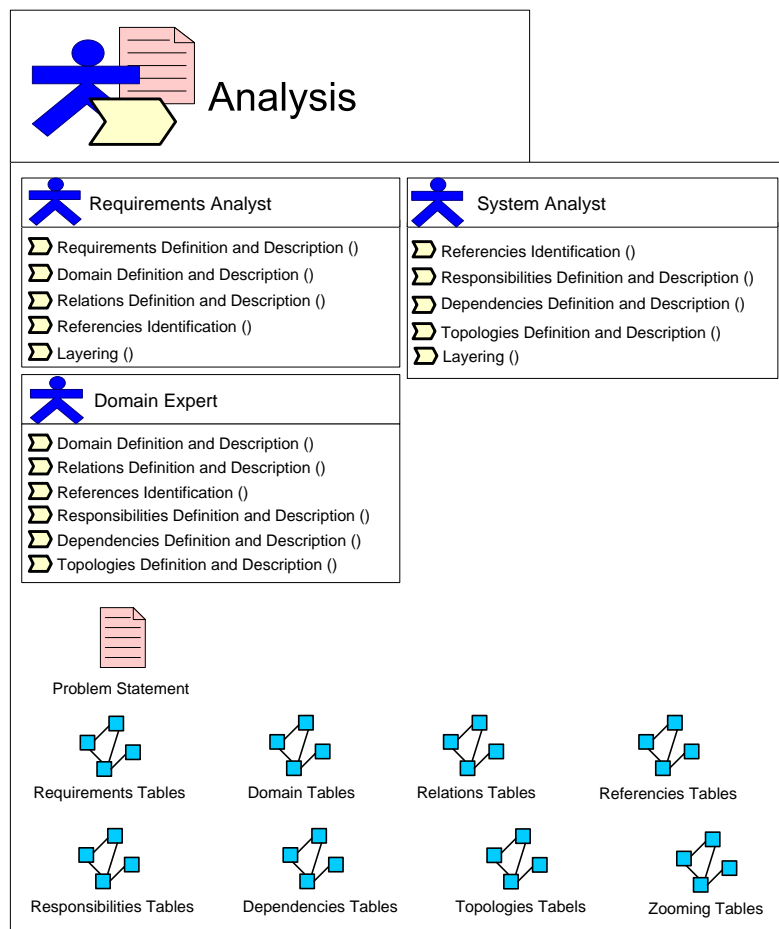


Figure 15.4: The Analysis Discipline

Figure 15.5 shows the Use-case diagram for the Analysis phase. The two WorkDefinitions of this phase – Requirements Analysis and Analysis – are decomposed in two sets of *activities* related to the WorkDefinitions by means of the *include* relations.

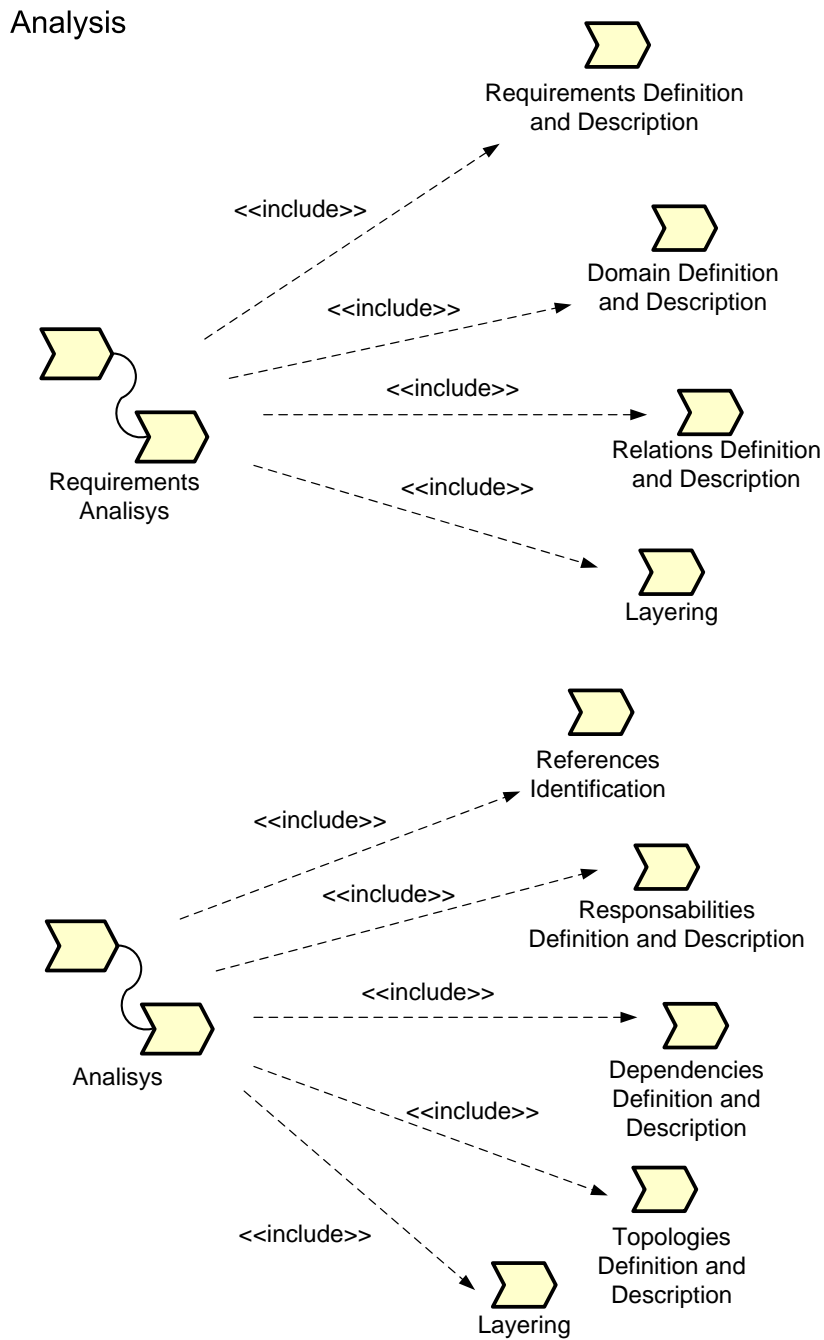


Figure 15.5: Use-case diagram of the Analysis phase

This discipline provides nine WorkProducts as showed in Figure 15.4: one text document and eight sets of relational tables. In particular Figure 15.6 depicts the process of the Analysis: this is composed by two WorkDefinitions (Requirements Analysis and Analysis) and their relative WorkProducts. The Analysis phase is an iterative process so when the Analysis step is terminated it is possible to re-start again the Requirements Analysis step. In addition the iteration could be done by means of the layering principle (see Section 14.3) both inside each step and between the steps (see Figures 15.8, 15.9 for the detail).

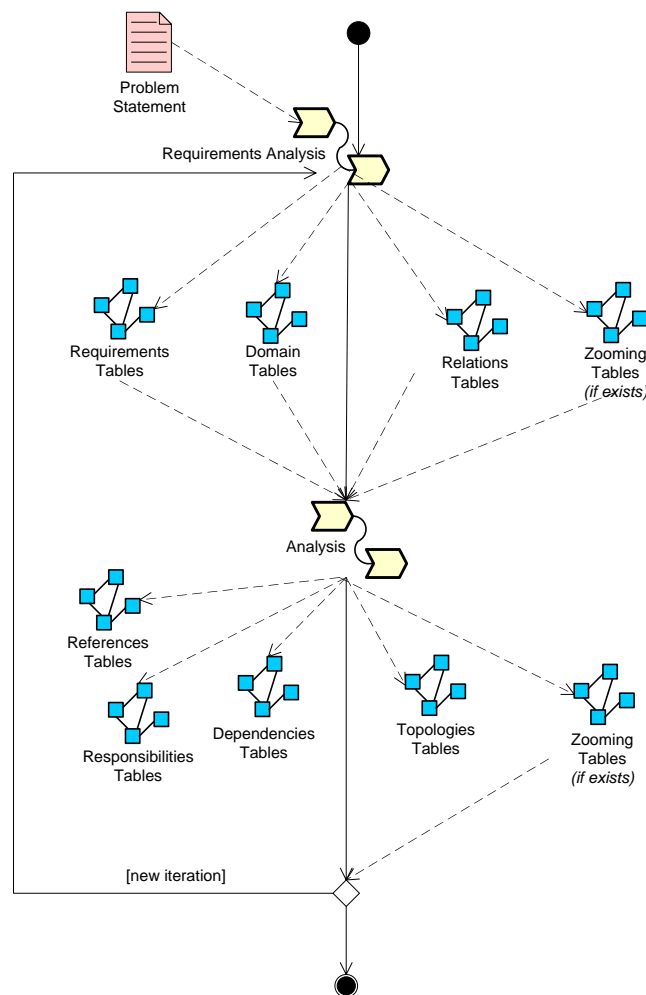


Figure 15.6: The Analysis phase described in terms of work definitions and work products

The actors are modelled by three *ProcessRole* instances: *Requirement Analyst*, *System Analyst* and *Domain Expert*. In particular:

- The *Requirement Analyst* is responsible (*performer*) of all the Requirements Anal-

ysis step activities and assists (*assistant*) the *System Analyst* during the Reference Identification activity.

- The *System Analyst* is responsible (*performer*) of all the Analysis step activities.
- The *Domain Expert* is the expert of the application domain, knows the legacy-systems and assists (*assistant*) the *Requirement Analyst* and the *System Analyst* during the modelling of the External-Environment.

Figure 15.7 shows the use-case diagram that models the relationships between the activities instances and the *ProcessRole* instances in SODA's Analysis phase [138].

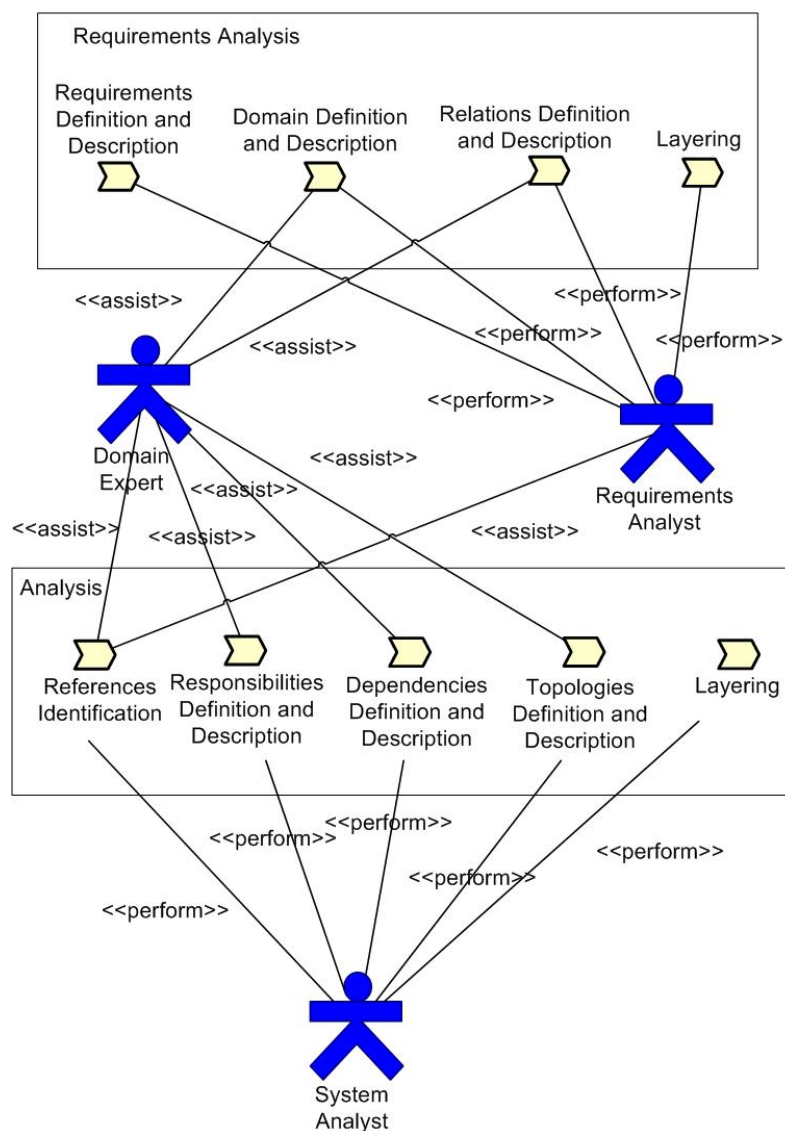


Figure 15.7: Work definition of the Analysis phase.

15.2.2 The Requirement Analysis step

Table 15.1 presents a summary of the activities of the Requirements Analysis steps that are presented in detail in Figure 15.8.

Activity	Description	Roles
Requirements Definition and Description	The input of this activity is the requirements specification (Problem Statement). The activity defines and describes the abstractions tied to the <i>requirement</i> concept. In particular the requirements are described and associated to their respective Actors.	Requirements Analyst (perform)
Domain Definition and Description	The input of this activity is the requirements specification (Problem Statement). The activity defines and describes the abstractions tied to the <i>external-environments</i> concept. In particular the legacy-systems are identified and described.	Requirements Analyst (perform) Domain Expert (assist)
Relation Definition and Description	The input of this activity is the requirements specification (Problem Statement). The activity puts in relation the abstractions of this steps. In particular the <i>relations</i> among requirements, among legacy-systems and among requirements and legacy systems are defined	Requirements Analyst (perform) Domain Expert (assist)
Layering	This activity represents the execution of both zooming and projection mechanisms	Requirements Analyst (perform)

Table 15.1: Requirements Analysis step activities

As it is possible to see in Figure 15.8, the *Requirements Definition and Description* and the *Requirements Definitions and Description* activities can be executed at the same time because they analyse different aspects of the problem domain. Indeed, the *Relation Definition and Description* activity can be executed when the abstract entities associated to at least one relation exist. Since this step is iterative, the *Relation Definition and Description* activity could start before the other two activities are finished. Obviously the layering can be applied during any activities of this step.

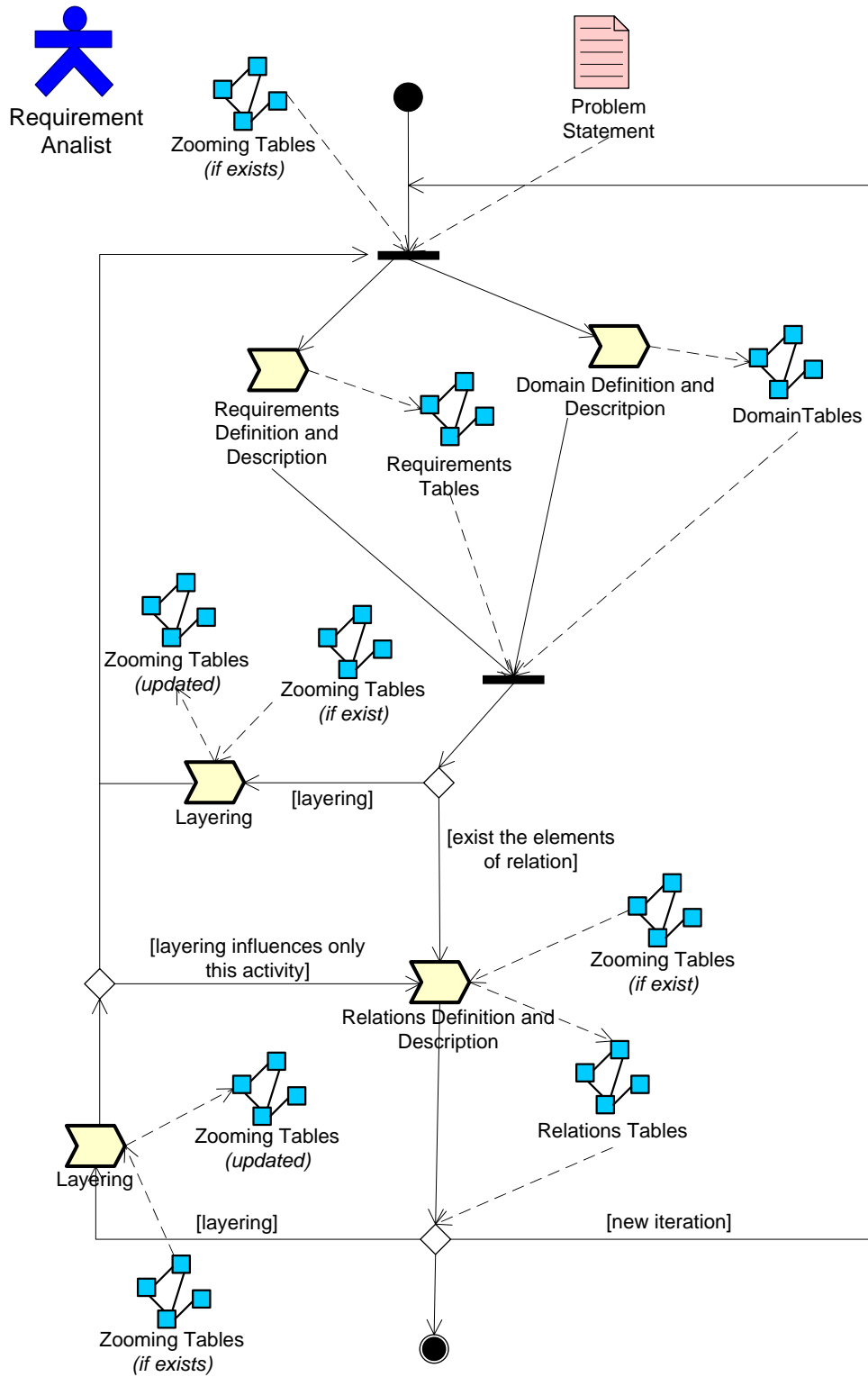


Figure 15.8: Description of the Requirements Analysis step work definition

15.2.3 The Analysis step

Table 15.2 presents a summary of the activities of the Analysis steps that are presented in detail in Figure 15.9.

Activity	Description	Roles
Reference Identification	This activity defines the relationships between the Requirements Analysis and Analysis steps. In particular the activity specifies the relation between the requirements and tasks, between the requirements and functions, between legacy-systems and functions, between legacy-systems and topologies, and between relations and dependency.	Requirements Analyst (assist) System Analyst (perform) Domain Expert (assist)
Responsibilities Definition and Description	The activity defines and describes the abstractions tied to the <i>responsibility</i> concept. In particular the sets of tasks and functions are described.	System Analyst (perform) Domain Expert (assist)
Dependencies Definition and Description	The activity puts in relation the abstractions of this steps. In particular the <i>dependencies</i> among tasks, among functions and among tasks and functions are defined.	System Analyst (perform) Domain Expert (assist)
Topologies Definition and Description	The activity defines and describes the topological constraints. In particular the constraints are listed and are putted in relation with tasks and functions	System Analyst (perform) Domain Expert (assist)
Layering	This activity represents the execution of both zooming and projection mechanisms	System Analyst (perform)

Table 15.2: Analysis step activities

The first activity in this step is the *Reference Identification* that represents the transition from the Requirements Analysis step to the Analysis step (Figure 15.9). The *Topology Definition and Description* can be executed at the same time of the *Responsibilities Definition and Description* and the *Dependencies Definition and Description* activities. It is possible to start the *Dependencies Definition and Description* activity before the *Responsibility Definition and Description* are finished because the *Dependencies Definition* can already begin when the entities associated to at least one dependency exist.

The *Topology Definition* activity could influence the *Responsibilities Definition and Description* and the *Dependencies Definition* activities, and in the same way the *Responsibilities Definition* could influence the *Topology Definition*. In the former case the table

modified by the *Topology Definition* activity must be provided to the *Responsibilities Definition* activity, and vice versa.

Obviously the layering can be applied during any activities of this step except the *References Identification*.

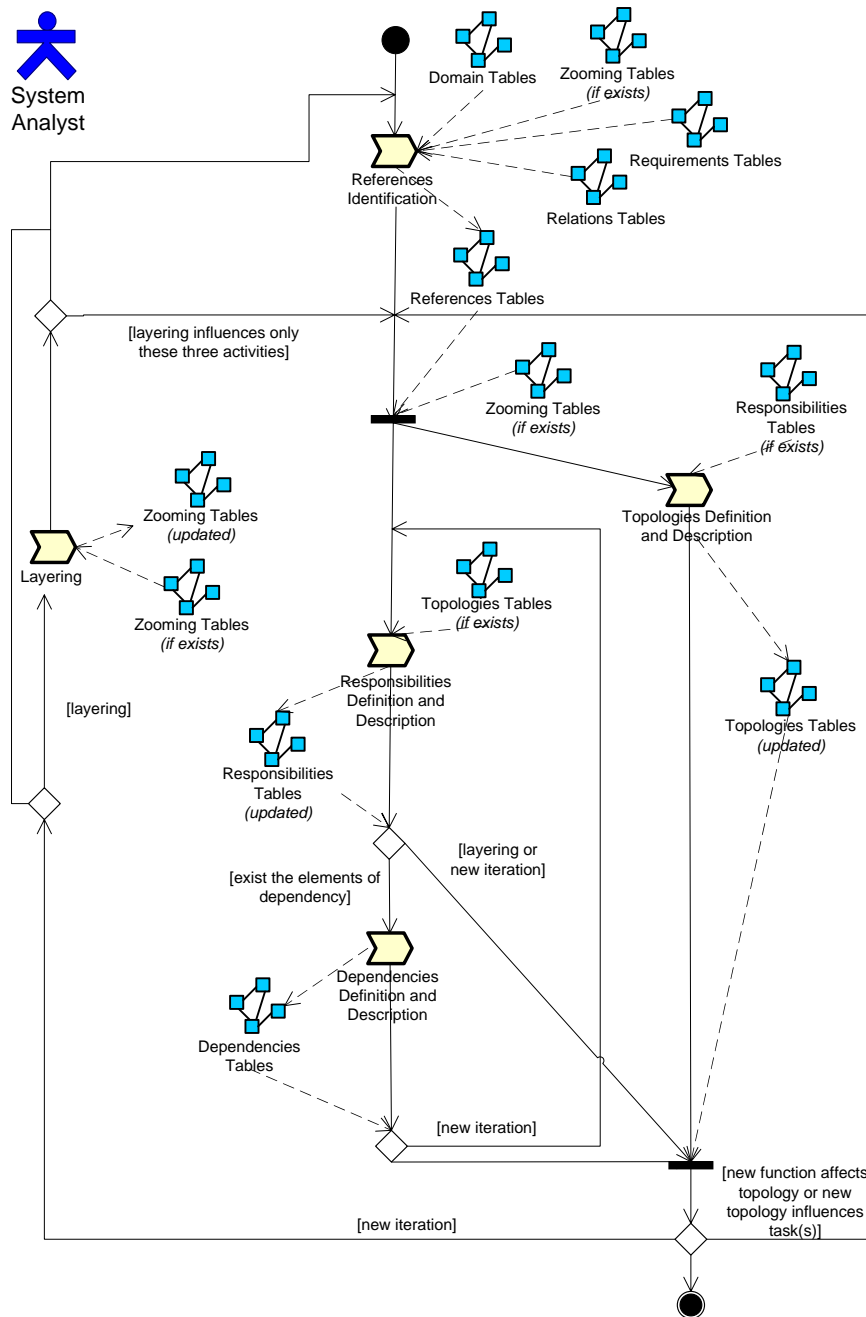


Figure 15.9: Description of the Analysis step work definition

15.2.4 The Analysis Model

Figure 15.10 shows the structure of the Analysis Model: the tables depicted in figure have already been explained in Section 14.4.

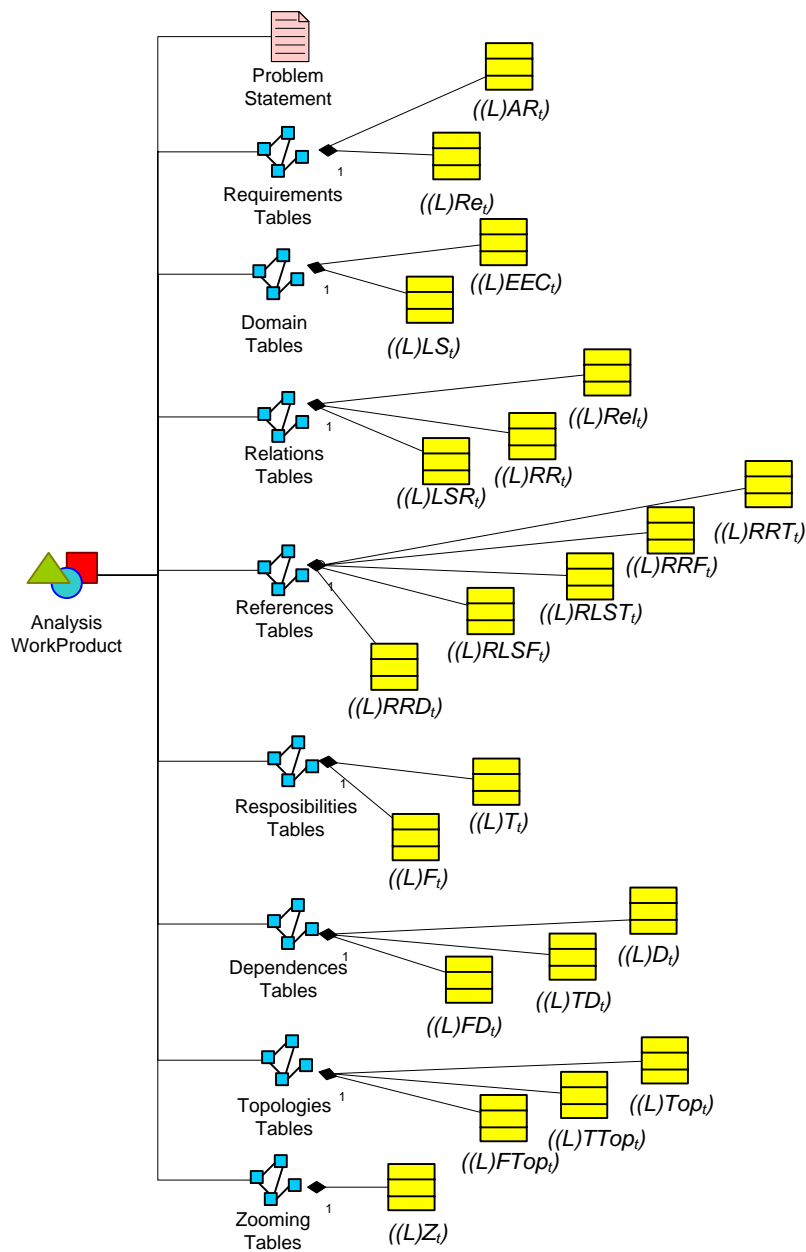


Figure 15.10: Structure of the Analysis Model

15.3 The Design Phase

This section presents the Design phase process. In particular Subsection 15.3.1 presents the Design discipline, Subsection 15.3.2 details the Architectural Design step, while Subsection 15.3.3 details the Detailed Design step. Finally Subsection 15.3.4 details the Design Model.

15.3.1 The Design Discipline

The *Design* discipline is characterised by four actors: the *Architectural Designer*, responsible for the activities of the *Architectural Design* step; the *System Designer*, responsible for the activities of the *Detailed Design* step; and the aforementioned *System Analyst* and *Domain Expert*, which support the two previous actors in performing their tasks.

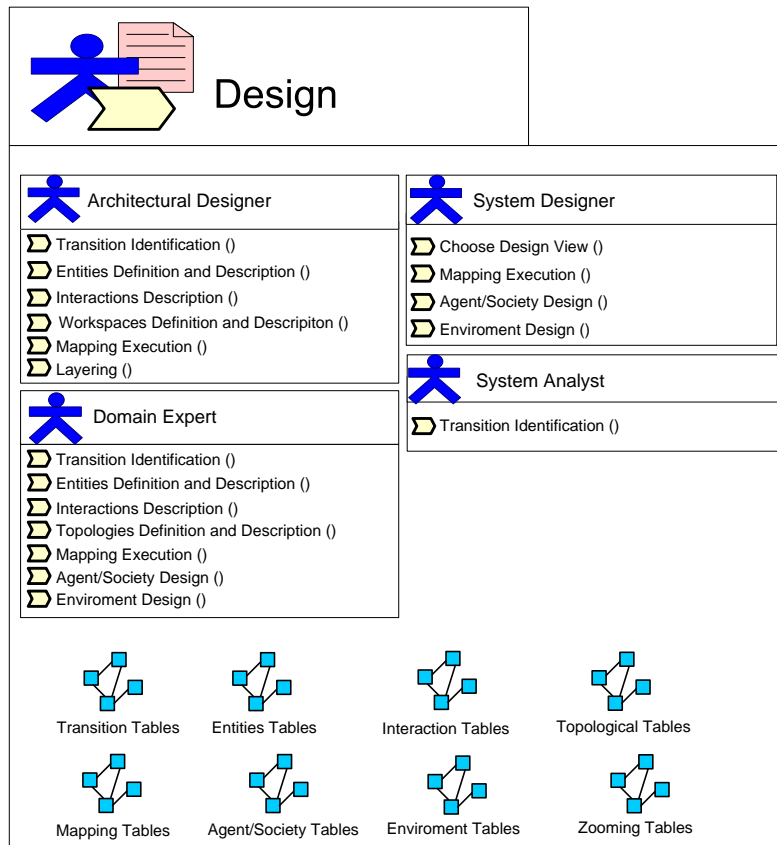


Figure 15.11: The Design Discipline

Figure 15.12 shows the Use-case diagram for the Design phase. The two WorkDefinitions of this phase – Architectural Design and Detailed Design – are decomposed in two

sets of *activities* related to the WorkDefinitions by means of the *include* relations.

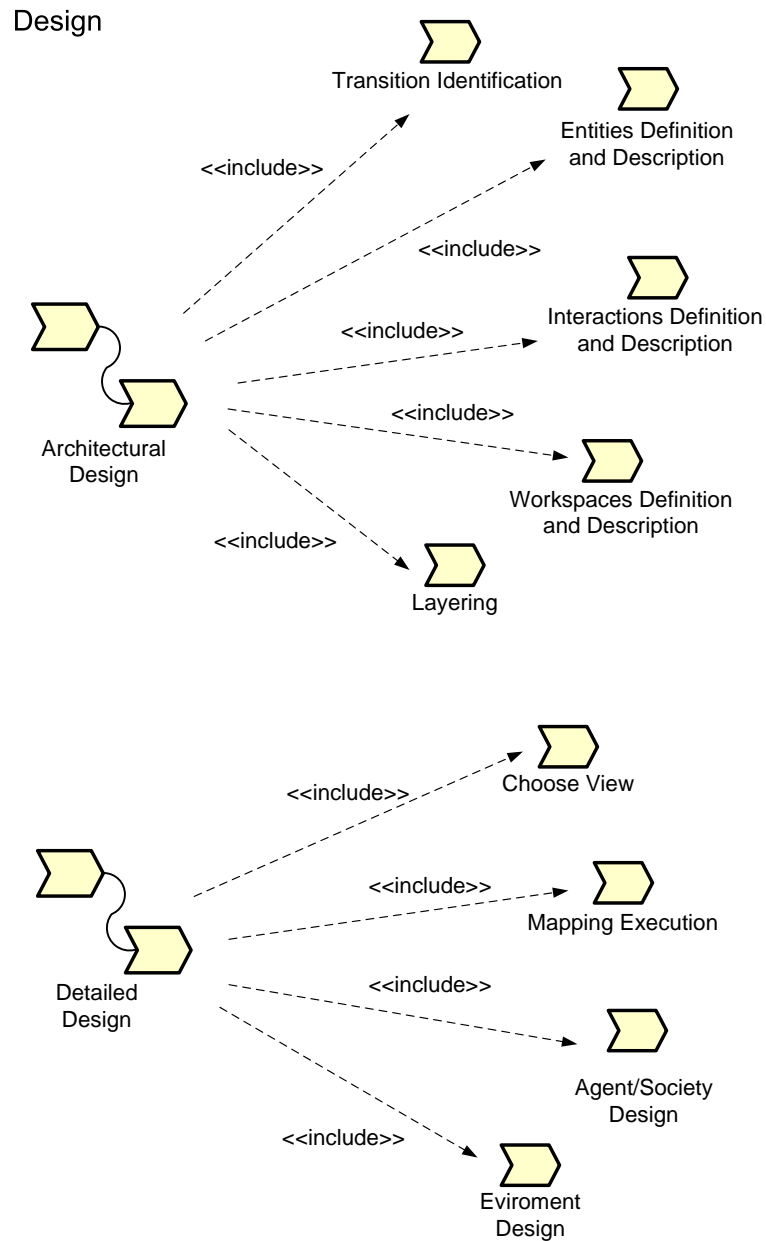


Figure 15.12: Use-case diagram of the Design phase

This discipline provides eleven WorkProducts as showed in Figure 15.11 that are sets of relational tables. In particular Figure 15.13 depicts the process of the Design: this is composed by the two WorkDefinitions (Architectural Design and Detailed Design) and their relative WorkProducts. The Design phase is an iterative process so when the Detailed

Design step is terminated it is possible to re-start again the Architectural Design step. In addition the iteration could be done by means of the layering principle (see Section 14.3) only for the Architectural Design step (see Figure 15.15 for the detail).

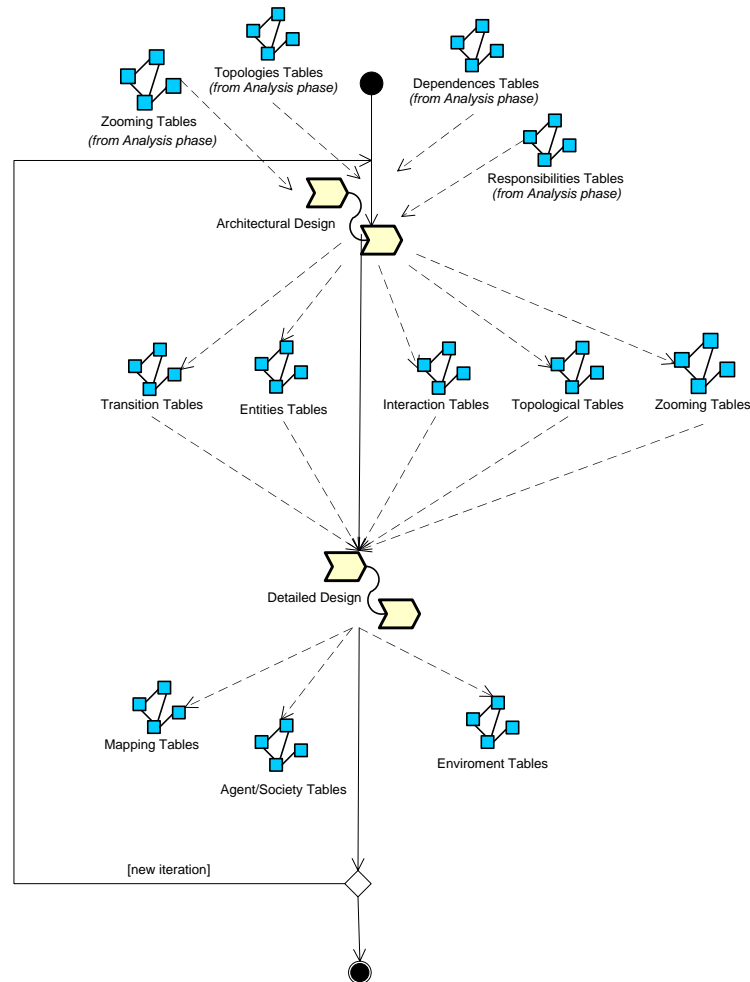


Figure 15.13: The Design phase described in terms of work definitions and work products

The actors are modelled by three *ProcessRole* instances: *Architectural Designer*, *System Designer*, *System Analyst* and *Domain Expert*. In particular:

- The *Architectural Designer* is responsible (*performer*) for the activities of the Architectural Design step and assists (*assistant*) the *System Designer* during the *Mapping Execution* activity.
- The *System Analyst* assists (*assistant*) the *Architectural Designer* during the *Transition Identification* activity.

- The *System Designer* is responsible (*performer*) for all the activities of the Detailed Design step.
- The *Domain Expert* is the expert of the application domain and assists (*assistant*) the *Architectural Designer* and the *System Designer* during the design of the Environment.

Figure 15.14 shows the Use-case diagram that models the relationships between the activities instances and the *ProcessRole* instances in SODA’s Design phase [138].

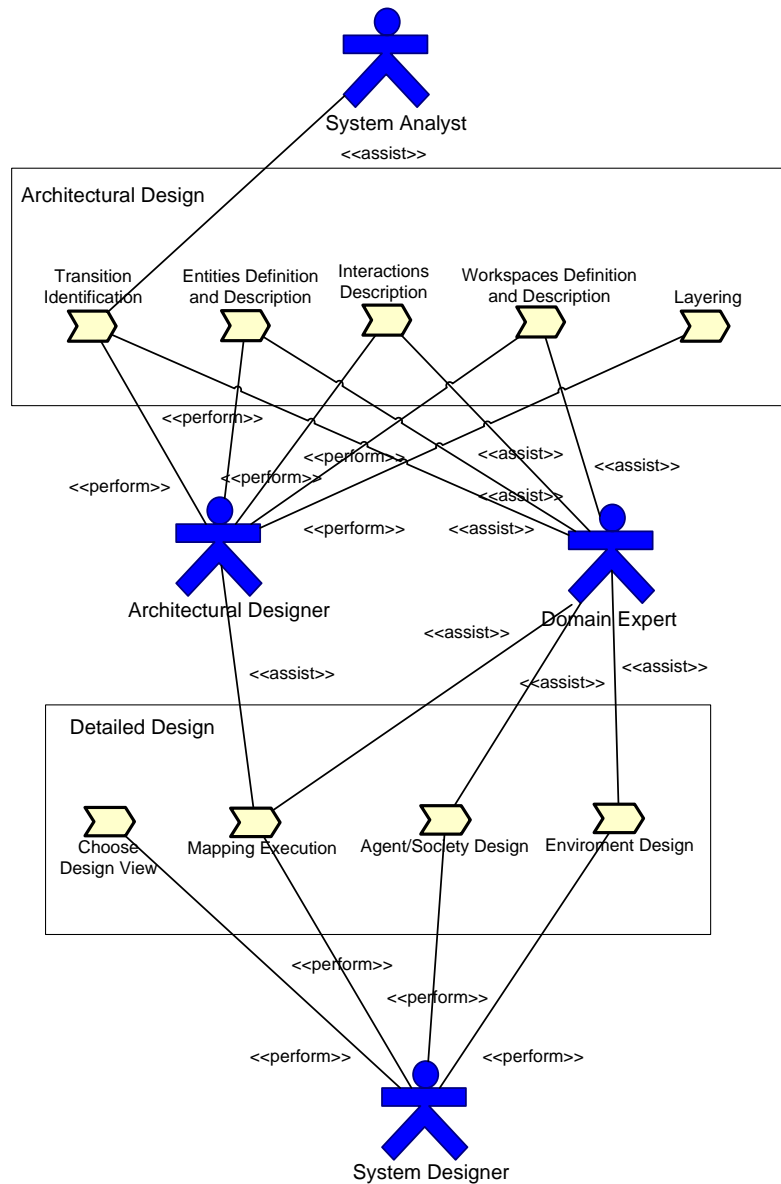


Figure 15.14: Work definition of the Design phase

15.3.2 The Architectural Design step

Table 15.3 presents a summary of the activities of the Analysis steps that are presented in detail in Figure 15.15.

Activity	Description	Roles
Transition Identification	This activity defines the relationships between the Analysis and Architectural Design steps. In particular the activity specifies the relation between the tasks and roles, between the functions and resources, between dependencies and interactions, between topologies and workspaces.	Architectural Designer (perform) System Analyst (assist) Domain Expert (assist)
Interaction Description	The activity defines and describes the abstractions tied to the <i>responsibility</i> concept. In particular the sets of tasks and functions are described.	Architectural Designer (perform) Domain Expert (assist)
Entities Definition and Description	The activity describes roles and resources. In particular the activity specifies the actions that each role is able to perform, and the operations provided by by the each resources	Architectural Designer (perform) Domain Expert (assist)
Workspaces Definition and Description	The activity defines and describes the workspaces. In particular the connection between workspaces of the same layer are showed, the resources are allocated in the workspaces, and the workspaces perceived by each role are illustrated	Architectural Designer (perform) Domain Expert (assist)
Layering	This activity represents the execution of both zooming and projection mechanisms	Architectural Designer (perform)

Table 15.3: Architectural Design step activities

The first activity in this step is the *Transition Identification* that represents the transition from the Analysis step to the Architectural Design step (Figure 15.15). The *Workspaces Definition and Description* can be executed at the same time of the *Entities Definition and Description* and The *Interaction Description* activities. It is possible to start the *Interaction Description* activity before the *Entities Definition and Description* are finished because the *Interaction Description* can already begin when the entities associated to at least one interaction exist. Obviously the layering can be applied during any activities of this step except the *Transition Identification*.

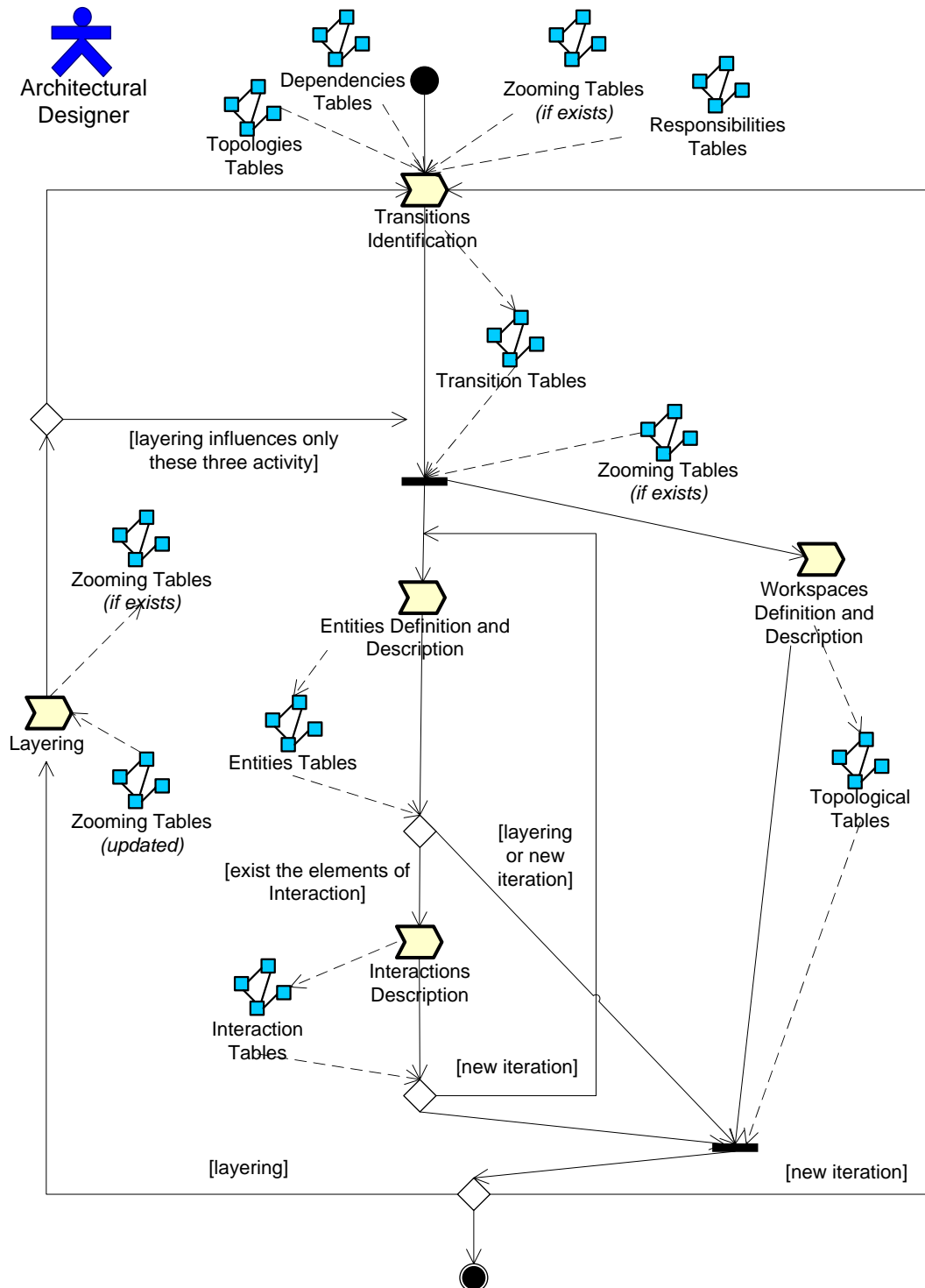


Figure 15.15: Description of the Architectural Design step work definition

15.3.3 The Detailed Design step

Table 15.4 presents a summary of the activities of the Analysis steps that are presented in detail in Figure 15.16.

Activity	Description	Roles
Choose Design View	The activity allows to choose the most suitable architecture by means the carving operation	System Designer (perform)
Mapping Execution	This activity defines the relationships between the Architectural Design and the Detailed Design steps. In particular the activity specifies the association between roles and agents, between the resources and artifacts, between interactions and artifacts	Architectural Designer (assist) System Designer (perform) Domain Expert (assist)
Agent/Society Design	The activity specifies the relations between agents and artifacts, agents and their relative societies (if the carving has produced societies), and which artifacts are associated to each society	System Designer (perform) Domain Expert (assist)
Environment Design	The activity specifies the artifacts' usage interfaces, the aggregations of artifact (if the carving has produced aggregates), and how artifacts are allocated to workspaces	System Designer (perform) Domain Expert (assist)

Table 15.4: Detailed Design step activities

The first activity in this step is the *Choose Design View* where the System Designer chooses the system architecture among the all the possible architectures identified in the Architectural Design step (Figure 15.16). The choice is done by means of the carving operation already explained in Subsection 14.5.3. Then it is possible to execute the *Mapping Execution* that maps the entities belonging to different layers in the Architectural Design step to the entities of the Detailed Design step. After that the *Agent/Society Design* and the *Environment Design* activities can be executed at the same time. At the end of this step if the architecture is not satisfactory is it possible to re-iterate the Detailed Design choosing a differen carving. Obviously the layering is forbidden in this step.

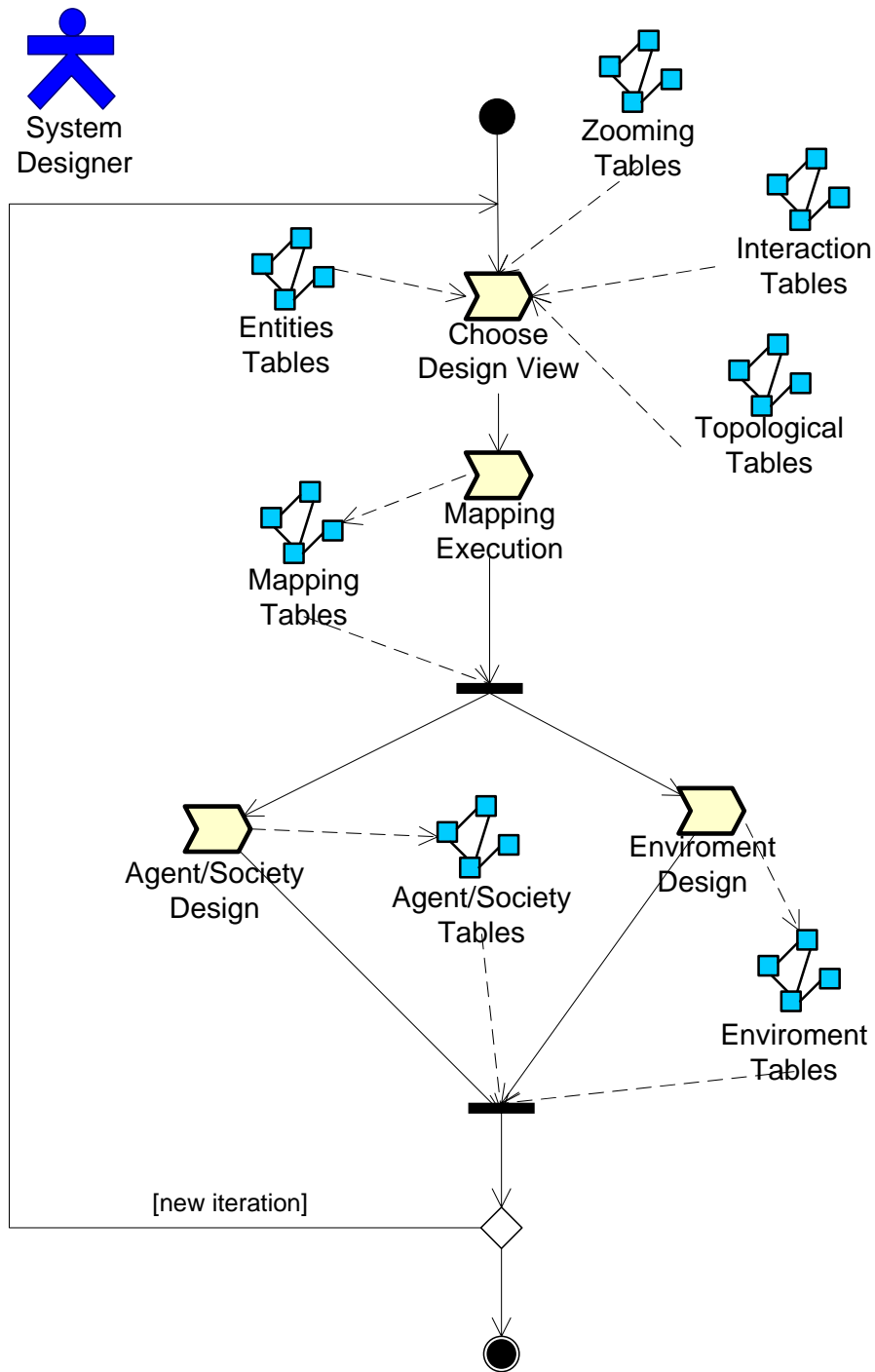


Figure 15.16: Description of the Detailed Design step work definition

15.3.4 The Design Model

Figure 15.17 shows the Design Model: the tables depicted in figure have already been explained in Section 14.5.

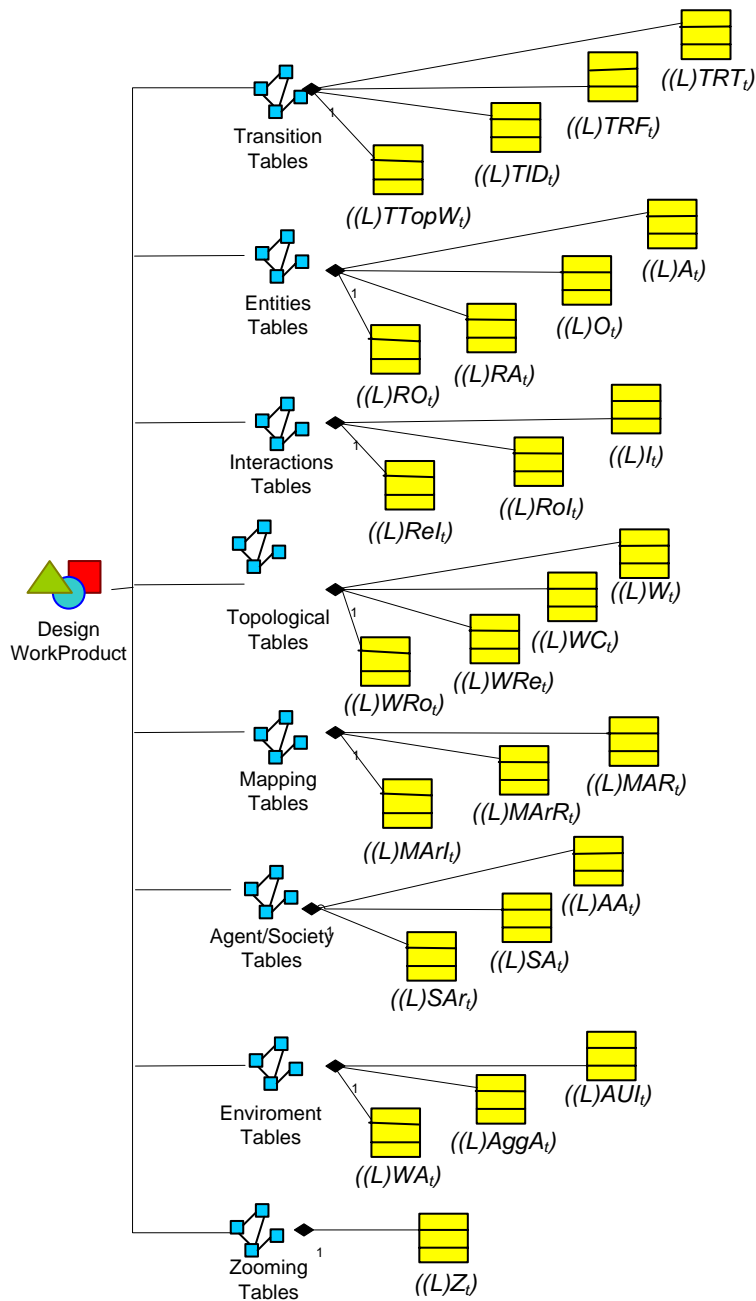


Figure 15.17: Structure of the Design Model

15.4 Summing up

Despite its origin in the object-oriented context, SPEM could be applied to the agent-oriented **SODA** process quite naturally, yet with some limits in expressiveness and readability. On the one side, in fact, the software development process and its phases are similar in any methodology, and mostly independent from the computational paradigm adopted. On the other, however, AO methodologies introduce a richer set of abstractions and mechanisms, which naturally lead to define a more articulated software development process: this sometimes stresses SPEM to its limits, showing its weakness in facing the increasing complexity—in particular, UML diagrams often become nearly unreadable when applied to AO methodologies.

In the specific case of **SODA**, for instance, the key issues of interaction and environment are apparently well modelled, instead the management of complexity presents some problems [138]. In fact, process iterations and applications of the layering principle are not easily captured by Activity diagrams—see Figure 15.15. Moreover, the *WorkProduct* elements are characterised by a unique symbol, which makes it very difficult to express their change of state during process iteration. Currently, the only means provided by SPEM to face this issue is the *Guidance* element, which can be used to express these aspects by barely adding descriptions (comments) to the Activity diagrams: of course, an *ad hoc* diagram specifically designed for process modelling would be a much better solution. Use-case diagrams, too, become very complex when modelling actors' activities (Figure 15.15). In fact, SPEM provides just only one symbol to represent two different types of association, *perform* and *assist*: so, it has to adopt different stereotypes – `<<perform>>` and `<<assist>>` – to tell one from the other. This choice makes the diagram unreadable if there are many activities and roles: again, enhancing SPEM to provide different symbols for modelling such associations would be a more effective and expressive solution.

Other methodologies modelled by the FIPA Methodology Technical Committee [121] apparently do not suffer from such limitations, mainly because they do not include mechanisms similar to the **SODA** layering principle: so, in such cases SPEM turns out to be able to capture the whole methodology process in quite an easy way. For instance, iterations of the activities in ADELFE can be modelled quite easily, and also the activity diagrams obtained from this methodology are clear, while **SODA** diagrams are nearly unreadable due to the considerably larger number of *WorkProducts*. However, in the overall the SPEM notation is not expressive enough for correctly modelling all the dynamics of the AO methodologies: for instance, the change in the *WorkProduct* status above is represented only in the label associated to the *WorkProduct* itself—there is no standard way to do this inside the notation.

Summing up, SPEM is apparently a good base to model AO software development processes, although the above limits make it difficult to represent the process effectively, leading to models that are sometimes uneasy to understand. So, an extension seems necessary in order to enable SPEM to overcome its limits in expressiveness and readability.

16

SODA & Infrastructures

This chapter discusses a method and some *guidelines* for correctly mapping the abstractions promoted by **SODA** design-level abstractions onto the infrastructural abstractions of **TuCSon** (Section 16.1), **CARTAgO** (Section 16.2) and **TOTA** (Section 16.3) already presented in Chapter 10: the abstractions used in the **SODA** analysis phase are left aside, as they would be too high-level with respect to infrastructures. In order to do this we apply the *meta-modelling* technique both to **SODA** and infrastructures and map the methodologies' abstractions onto the infrastructures' abstractions at the meta-model level. This method allows to fill the gap between methodologies and infrastructures that can lead to *dangerous inconsistencies* between the design and the actual implementation of the system (Subsection 3.3.1). These are the consequences of the use of concepts and abstractions in the analysis and design stages which are different from those used to deploy and implement the system: methodologies can be exploited precisely to overcome such a gap, however monitoring and incrementally evolving 24/24-7/7 systems call for design-to-deployment abstractions and keeping the abstractions alive.

Among the many AO infrastructures available in the literature, we choose **TuCSon** and **TOTA** because interaction – and coordination, in particular – is at the core of both infrastructures, in the same way as in **SODA**. Finally we choose **CARTAgO** because it is the only infrastructure that natively supports the concepts of artifacts.

Such infrastructures are then compared so as to evaluate their support to **SODA** abstractions (Section 16.4). A summary follows in Section 16.5.

16.1 From SODA to TuCSon

Since **SODA** is defined on top of the A&A meta-model, the first step is to define how agents and artifacts can be represented as **TuCSon** abstractions. In order to simplify the presentation the **TuCSon** meta-model from Chapter 10 is reported in Figure 16.1.

Mapping the agent notion is straightforward, given **TuCSon** native support for this concept: so there is a one-to-one mapping between the **SODA** agent abstraction and the **TuCSon** one. However, the concept of agent action, explicitly considered in the **SODA** meta-model, is more or less reduced to the notion of coordination primitives, as performed

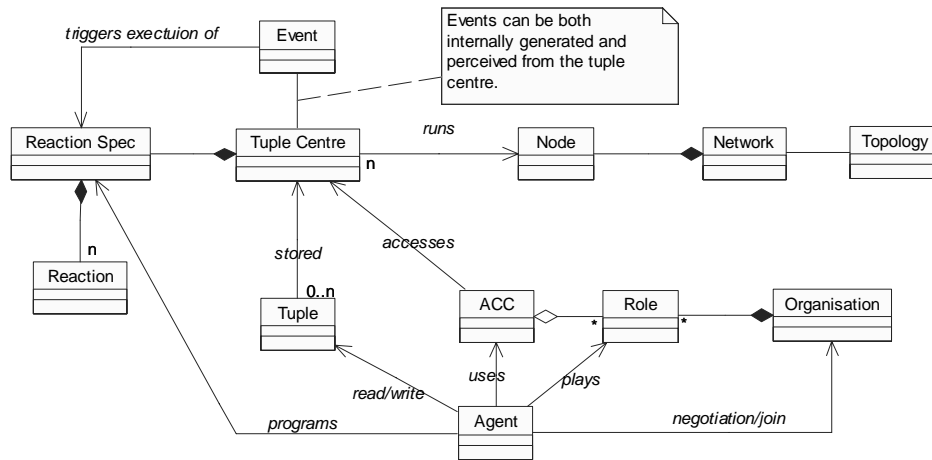


Figure 16.1: TuCSoN Meta-model

by agents. The coordination primitives supported by **TuCSoN** are: $out(t)$ is used to put a passive tuple t in the tuple centre, $in(t)$ retrieves a passive tuple t from the tuple centre, $rd(t)$ retrieves a copy of t from the tuple centre (i.e., t is still there) and $eval(p)$ puts an active tuple p (i.e., a process) in the tuple centre.

Mapping the artifact notion, instead, is less obvious, as **SODA** defines three different artifact types – social, individual and environmental artifacts – each requiring its own mapping. With respect to this issue, **TuCSoN** tuple centres can be seen as a special case of social artifacts: they mediate and govern agent interaction by encapsulating the laws of agent coordination. Such coordination laws, expressed in terms of reactions to interaction events, are well suited to mapping **SODA** interactions—i.e., the rules that enable and bound the entities’ behaviour. In turn, the notion of individual artifact can be mapped onto the **TuCSoN** ACC concept, since its purpose is precisely to represent the interface of an agent towards the environment [215]. In fact, agents ask for an ACC specifying the roles to be activated: the ACC is then negotiated with the infrastructure as the agent joins the MAS organisation. If the negotiation is successful, the ACC is created and released to the agent, which, henceforth, exploits it to access the MAS services: the ACC redirects the agent invocations to the other artifacts in the environment. Finally, the environmental artifact is not natively supported by **TuCSoN**, so it must be developed if/when needed. Also, the notion of artifact operation is reduced here to the notion of tuple centre operation, and has not the generality required. The general form for any admissible **TuCSoN** operation is $tc?operation(tuple)$ asking tuple centre tc to perform $operation$ using $tuple$.

Given that, link operations through tuple centres are the way in which **TuCSoN** “artifacts” are somehow composed and represent the **SODA** aggregates.

Widening the view, the organisation concept provided by **TuCSoN** is well suited to

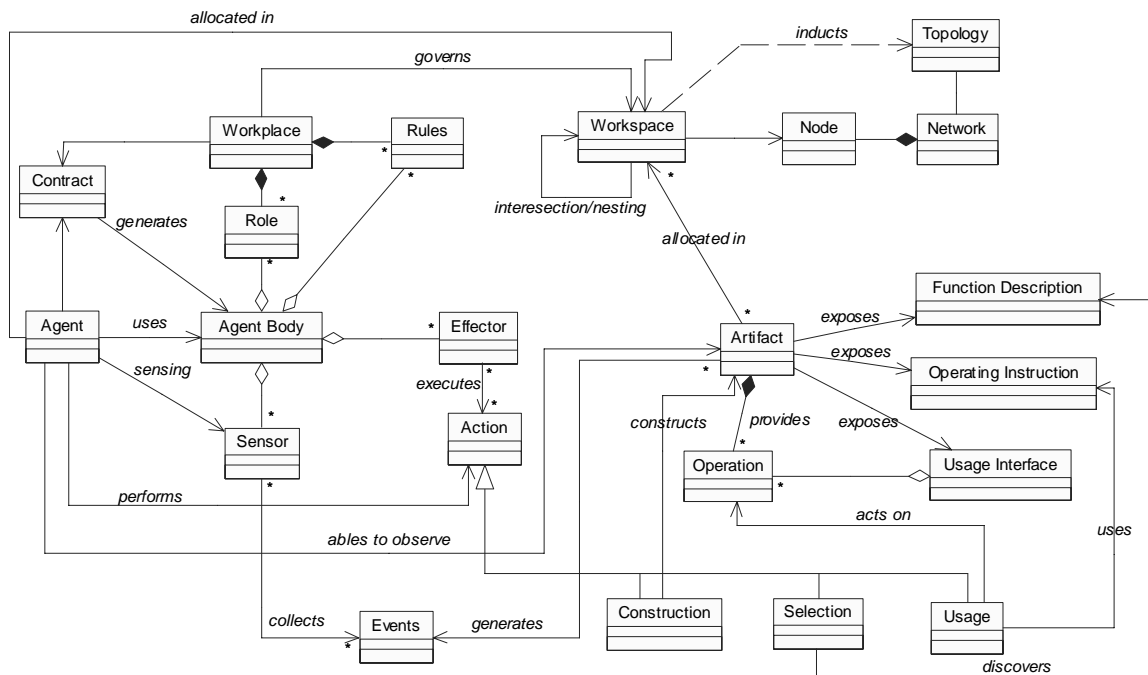


Figure 16.2: CArtAgO Meta-model

represent **SODA** societies, in the same way as the **TuCSoN** role concept can represent the **SODA** role notion. From the topological viewpoint, the **SODA** notion of workspace may be mapped onto the **TuCSoN** node concept, which, indeed, represents an open set of agents, tuple centres and ACCs; as a consequent step, the **TuCSoN** network can be used to map **SODA** environments. On the other hand, workspace connection, as introduced by **SODA**, has no mapping in **TuCSoN**, so it should be developed in an ad hoc way when needed.

16.2 From SODA to CArtAgO

CArtAgO and **SODA** share the same root in the A&A meta-model: so, quite expectedly, **CArtAgO** abstractions can easily support all **SODA** concepts. In order to simplify the presentation the **CArtAgO** meta-model from Chapter 10 is reported in Figure 16.2. **CArtAgO** provides the artifact notion as a first-class abstraction, which can be used and easily specialised in social and environmental artifacts according to the developer's needs. Therefore, unlike **TuCSoN**, the **SODA** notion of artifact operation is directly mapped onto the operation abstractions supported by **CArtAgO**; the same holds for the **SODA** agent action supported by **CArtAgO**'s action. Moreover, individual artifacts can be more specifically mapped to the **CArtAgO** agent body abstraction, instead of using the generic

artifact notion.

Aggregate can also be easily realised, thanks to the *linkability* property [152] natively supported by **C**ArtAgO artifacts to scale up with environment complexity. So, an artifact can be conceived and implemented as a composition of linked, possibly non-distributed, artifacts, or, conversely, a set of linked artifacts, scattered through a number of different physical locations, can be seen altogether as a single distributed artifact.

In addition, **SODA** organisational structure, which is defined in terms of roles and societies, can be easily translated on **C**ArtAgO roles and workplaces. This makes it possible to capture the **SODA** interaction concept in a straightforward way: in fact, interactions in **SODA** are aimed at enabling and constraining agent behaviour, which is precisely what the workplace rules and contract do—a **C**ArtAgO agent may or may not have permission to use some artifacts or to execute some specific operations on some specific artifacts depending on the role(s) that the agent itself is playing inside the workplace.

Finally, the **C**ArtAgO workspace concept can be directly used to map the **SODA** workspace concept, in the same way as the **C**ArtAgO workspace nesting supports the **SODA** workspace connection. The **C**ArtAgO abstractions of node, network and topology can be used to represent the **SODA** environment, too.

16.3 From SODA to TOTA

TOTA provides native support to the agent concept, while the artifact concept is supported only in the case of social artifacts. So, **SODA** agents can be directly mapped onto **TOTA** agents, while social artifacts are mapped onto **TOTA** tuple spaces. In order to simplify the presentation the **TOTA** meta-model from Chapter 10 is reported in Figure 16.3. Unlike tuple centres, tuple spaces provide only a fixed coordination service: so, they are unable to support the **SODA** interaction concept. However, **SODA** social rules can be mapped onto the maintenance rule and the propagation rule associated to **TOTA** distributed tuples, exploiting the fact that propagation rules determine how tuples propagate through the network, and maintenance rules determine how the tuple distributed structure reacts to environment events. Of course, this mapping is less straightforward than in **TuCSon** (whose reactions map the **SODA** interaction concept directly): indeed, a set of many tuples must be used to describe a single **SODA** interaction—each tuple representing one propagation and one maintenance rule. As a side effect of this one-to-many mapping, maintaining coherency is quite a hard task, and the rules/interaction mapping can often be very dispersed.

From the topology viewpoint, the **TOTA** node concept maps to the **SODA** workspace concept: each node holds references to a limited set of neighbour nodes, and neighbourhood relations express the network topology. Such inter-node relations can be exploited also to provide an abstraction for mapping to the **SODA** workspace connection concept. Finally, the **TOTA** network concept maps to the **SODA** environment, too. All the others

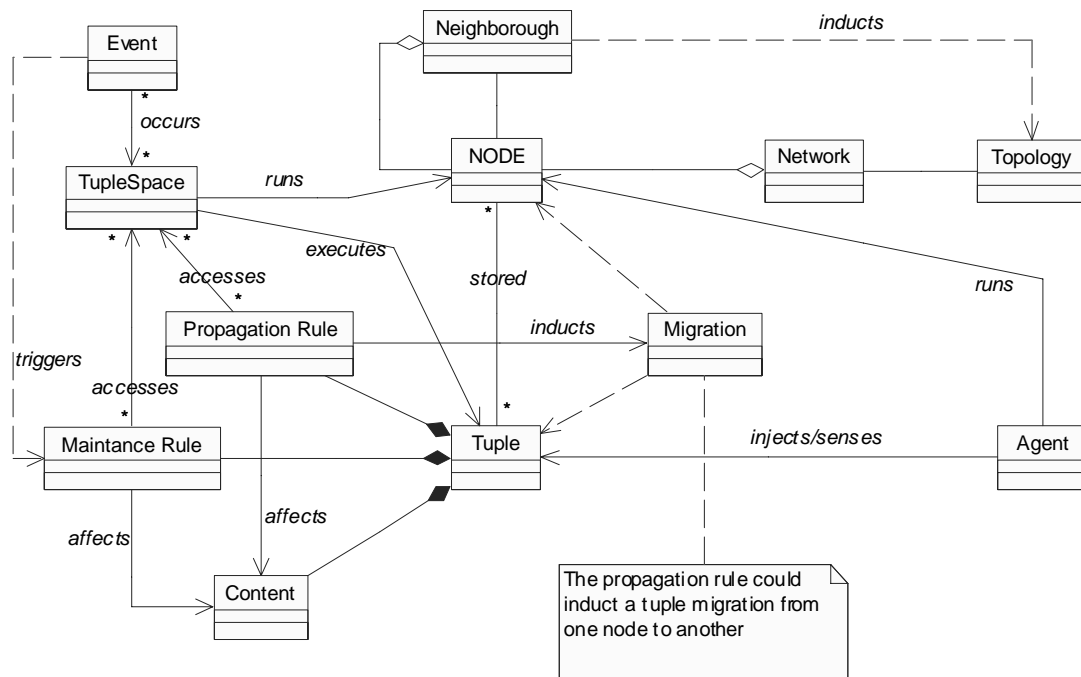


Figure 16.3: TOTA Meta-model

SODA concept are not natively supported by TOTA, and should therefore be developed in an *ad hoc* way when needed.

16.4 Discussion

Figure 16.4 highlights the SODA abstractions that are supported natively from each of the three infrastructures. The agent and resource abstractions are both omitted—the first because it is explicitly supported by each infrastructure, the latter for the opposite reason.

Quite expectedly, CArtAgO provides the best support for SODA design abstractions, as they are both rooted in the A&A meta-model: in particular, both consider the environment as the key element, adopt artifacts as their basic building blocks for modelling the environment resources, and workspaces for structuring the environment. Moreover, both SODA and CArtAgO support the MAS organisational structure by explicitly enabling the specification of social rules.

TuCSon and TOTA, instead, provide support for fewer SODA abstractions: so, the developer needs to implement by himself the abstractions which are not supported by the infrastructure natively. In particular, none of the two infrastructures supports environmental artifacts, while both support social artifacts: this is not surprising, since they take both inspiration from coordination models, where interaction is typically mediated

SODA	TuCSon	CARTAgO	TOTA
<i>Role</i>	<i>Role</i>	<i>Role</i>	-
<i>Action</i>	<i>Coordination Primitive</i>	<i>Action</i>	-
<i>Interaction</i>	<i>Reaction Specification</i> <i>Reaction</i>	<i>Rules</i> <i>Contract</i>	<i>Maintenance Rule</i> <i>Propagation Rule</i>
<i>Operation</i>	<i>Tuple Centre Operation</i>	<i>Operation</i>	-
<i>(Social) Artifact</i>	<i>Tuple Centre</i>	<i>Artifact</i>	<i>Tuple Space, Tuples</i>
<i>(Individual) Artifact</i>	<i>ACC</i>	<i>Agent Body</i>	-
<i>(Environmental) Artifact</i>	-	<i>Artifact</i>	-
<i>Aggregate</i>	<i>Linked Tuple Centres</i>	<i>Artifact</i>	-
<i>Society</i>	<i>Organisation</i>	<i>Workplace</i>	-
<i>Workspace</i>	<i>Node</i>	<i>Workspace</i>	<i>Node</i>
<i>Workspace Connection</i>	-	<i>Workspace Nesting</i>	<i>Neighborhood</i>

Figure 16.4: Abstractions Mapping

by some coordination media [71] (like a tuple space or a tuple centre) that could be easily seen as a special case of social artifact. Individual artifacts, in turn, find their counterpart only in **TuCSon**—namely, in the ACC abstraction. Moreover, **SODA** interaction abstraction, which represents the rules that enable and shape agent behaviour, can be expressed directly by **TuCSon** reactions, and indirectly via **TOTA** maintenance and propagation rules. Finally, as far as the organisational structure of the MAS is concerned, **TuCSon** provides explicit abstractions such as organisation, role and ACC; on the other hand, **TOTA** does not provide any support for this issue yet, so the developer must provide for managing the MAS organisations on his/her own.

16.5 Summing up

This chapter comparatively analysed the meta-models of the **SODA** methodology and of **CARTAgO**, **TuCSon**, and **TOTA** infrastructures, with double purpose of (a) providing a method for filling the gap between the design and implementation phases, and (b) evaluating the quality of the mapping of **SODA** concepts onto infrastructural abstractions in terms of naturalness, clearness, and directness of the mapping. The method presented in this chapter exploits the meta-models as a tool for mapping between methodologies and infrastructure.

As for the methodologies, a meta-model enables the representation of an AO infrastructure, exposing its inner architecture and behaviour in a technology-independent way. A well-defined meta-model should represent both the static and the dynamic aspects of an infrastructure, so we can exploit infrastructure meta-models to associate each methodology concept to some suitable infrastructural abstraction.

17

Case Study

The complexity of today's software systems makes the scientific study of computational models, technologies and methodologies a non-trivial issue—including, in particular, the choice of significant case studies. In fact, the necessarily limited size of presentations and articles in scientific conferences and journals – as well as the limited attention span of reviewers, one may add – typically makes researchers lean towards application scenarios that are small enough to be discussed and understood in a few pages. However, this often undermines the general validity of the approaches presented when applied to complex computational systems – that is, precisely those they were meant to – in spite of the usual claims by authors.

This is exactly what makes the Conference Management System (CMS) scenario such an appealing one to put technologies and methodologies to test, and a so-frequently chosen one in the literature [30, 56, 117, 191, 194]:

- it is apparently simple—actually, the goal of the CMS is simple and easy to understand, that is, revising and selecting scientific papers for conference presentation and proceedings;
- yet, it is quite an articulated application scenario, characterised by a wide range of interaction aspects (coordination, organisation, and security problems), as well as by frequent requirements for structural evolution and changes;
- it is usually known in most of its fine-print details and subtle nuances by readers / reviewers, who have often taken part in such a kind of process;
- it has been used as a testbed for technologies and methodologies in the literature, and as such it has been described and analysed in depth several times.

These features allow authors to concentrate on the specific aspects of interest, avoiding the description of most well-known issues while preserving the effectiveness of the application scenario in representing complex systems.

The goal of this chapter is twofold. On the one hand, the aim is to demonstrate the power of AO methodologies, in particular when based on an expressive meta-model

like the A&A one. On the other hand, we mean to prove the effectiveness of **SODA**, in particular when extended with layering mechanisms to handle the engineering of complex computational systems.

So, the chapter is structured as follows. Section 17.1 introduces the conference management application scenario: in particular, Section 17.2 discusses CMS in the literature and emphasises some key lessons to be learned, while Section 17.3 presents the CMS scenario from the MAS engineering viewpoint, and gives the fundamentals of the A&A meta-model. Next, Section 17.4 provides details about the design and modelling of conference management in **SODA**, while Section 17.6 reports on the benefits of the methodological approach presented.

17.1 Conference Management Systems

The management of scientific conferences is an interesting case study [30, 229] in that it involves several aspects, from the main organisation issues to paper submission and peer review, which are typically performed by a number of people distributed over the world who exploit the Internet as the infrastructure for communication and cooperation. Setting up and running a conference is a multi-phase process, involving several individuals and groups. During the submission phase, authors send papers, and are informed that their papers have been received and have been assigned a submission number. In the review phase, the program committee (PC) handles the paper review, contacting potential referees and asking them to review one or more papers. Eventually, reviews come in and are used to decide about the acceptance / rejection of the submissions. There are various ways to distribute the papers: *i)* the PC scans all papers and distributes them according to personal preferences; *ii)* there can be an automatic match according to keywords; *iii)* titles and abstracts may be broadcast, then a bidding mechanism can be implemented; *iv)* the PC-chair decides to distribute the papers based on the expertise of the various PC-members. The situation gets more complex if, as it is often the case, PC-members are allowed to submit papers: in this case, one must prevent them from accessing (or even just inferring) information about their own submissions. In the final phase, authors are notified of these decisions and, in case of acceptance, are asked to produce a revised version of their paper. The publisher then collects these final versions and prints the proceedings.

In addition, the overall structure of the conference organisation may dramatically vary from year to year, for two main reasons [28]. First, the involved organisers often change every year, thus inducing changes in the organisation due to their personal attitudes and opinions. Second, the conference topics and the effectiveness of the conference advertising may strongly affect the number of submitted papers, possibly forcing to change the structure of the management process in order to keep it manageable while scaling with the conference dimension. This is particularly true for the reviewing process, which involves

a large number of actors, with different duties and variously interacting with each other.

17.2 CMS in the literature

Conference management has been considered in several research areas, each highlighting some specific problem issues—from coordination to workflow management, from mobile computing to the organisational structure.

CMS: The Coordination Viewpoint. Ciancarini [30] is one of the first authors to study conference management from the viewpoint of coordination. There, this case study is considered as an example of a real life application with prominent coordination aspects, since it involves a set of coordination activities but also requires coordination for its own workflow. In particular, managing the conference workflow is expressed as managing the dependencies of the activities needed to produce the proceedings of a scientific conference. Scutellà [194], instead, presents a simulation based on MANIFOLD, whose scope is the automation of the paper submission process and the distribution of papers to referees. Since MANIFOLD is a strongly-typed, block-structured, event-driven coordination language with asynchronous communication, aimed at enforcing the separation between computation and communication concerns, the proposed solution is structured into a hierarchy of processes: a PC-chair (the coordinator), and a set of PC-members (the coordinated processes). Event-based methods are used as the coordination means, while the information flow is enacted using point-to-point channels. Analogously, in [131], the conference management problem is used to pragmatically validate Oikos_adtl, a specification language for distributed systems based on asynchronous communication via remote writing, which makes it possible to express the global properties of coordination templates. There, however, the only addressed issue is the allocation of submitted papers to PC members for reviewing. As a last example, Rossi and Vitali [187] discuss a solution based on PageSpace, an architecture for Web-based applications where autonomous agents can perform their tasks regardless of their physical positions. This architecture is instantiated with a coordination language offering mobility facilities, and follows the Multi-User Dungeon metaphor – a cooperative interactive environment shared by several people to socialise and interact. The virtual space is partitioned in rooms, which contain users and items: users can move from a room to another, and can interact with the items and with the other users in the current room. Accordingly, in the implemented system, *ConfManager*, submitted papers are stored in rooms, while authors, reviewers, and PC-Members are represented by avatars, exploiting both synchronous (online) and asynchronous (e-mail) communication interactions.

CMS: The Organisation Viewpoint. Organisational issues are central in [229], where the conference management system is structured as a number of different MAS organisations, one for each phase of the conference management process. In each organisation, the corresponding MAS can be viewed as a set of agents associated to the people

involved in the process (authors, PC-chair, PC-members, reviewers), who represent the active part of the system. In addition, Cernuzzi and Zambonelli [28] argue that, although the process of designing a MAS to support the organisation of a conference may conceptually appear quite simple, the most critical issue is the extreme variety of real-world organisations, which often change from year to year. This aspect not only impacts the design of the MAS in terms of the types and roles of the involved agents, of the organisational structure and of inter-agent interactions: it also calls for adaptive MAS and for a design approach coherent with a “design-for-change” perspective.

CMS: The Mobility Viewpoint. From a totally different viewpoint, Cardelli exploits conference management as a “challenge for any wide area language to demonstrate its usability” [22], highlighting the issues of code mobility. Another example can be found in [56], where Mobile Maude – an extension of Maude supporting mobile computation – is used to represent and specify ambitious wide-area applications. A conference reviewing system is then used as a testbed for executable Maude specifications, in particular with the aim of identifying possible bugs and experimenting with different alternatives.

CMS: The Systems Viewpoint. Implementations of conference management systems can be found in [117, 191], which describe the electronic management systems developed, respectively, for the 4th and 5th International World Wide Web Conferences, and for the 7th International Conference on Concurrency Theory and the 24th International Colloquium on Automata, Languages, and Programming. Other applications are available for conference management [57, 136, 34, 218, 161]: they all try to automate the long or repetitive tasks (possibly using sophisticated algorithms, as in the case of [136]), such as assigning papers to reviewers and collecting paper reviews. Some applications also try to face the most common security issues, adding some form of access control (usually, username and password) to paper information, so that each author can access only the information about his/her own paper.

CMS: Lessons Learned and Requirements. Conference management is popular in the literature because it is at the same time easy-to-understand, real-life, yet complex enough in terms of coordination, organisation, security, and scalability to be an effective testbed for a variety of programming paradigms, methodologies and technologies. However, such a versatility has sometimes led authors to use it just to model some specific aspects – the ones which are best suited to their own approach or technology (paper assignment to reviewers, collecting paper reviews, mobility. . .) – rather than to actually design and engineer a real application.

For this purpose, several requirements should be met:

- the whole set of actors and interaction should be modelled – not just the “part of interest” – using suitable high-level metaphors and adequate software engineering methodologies; such metaphors should have their counterpart in the selected implementation technology both at design time and run time.
- it should be possible to specify any kind of coordination pattern, as required to

suitably coordinate several activities (paper assignment, peer review, discussions among PC members, collecting reviews, etc.);

- adequate organisational abstractions should be available in order to express organisational constraints, environmental aspects and security issues as needed; these may include, for instance, concepts such as roles, workspaces, etc.
- last but not least, the selected software infrastructure should be based on open architectures, so as to support dynamic evolution and meet the new requirements in response to possible modifications of the organisational structure.

17.3 CMS & Agent-Oriented Approach

Conference management is popular in the literature because it involves interaction among loosely-coupled peers spread over the world, thus intrinsically stressing the issue of suitably coordinating several activities. The agent-oriented approach is seemingly good in managing these scenarios: agents are autonomous and proactive, and can engage in elaborate interactions both with other agents and with the environment. This is why it appears natural to think of MAS for the design of complex software systems like CMS.

CMS & Multi-Agent Systems The MAS approach enables designers to face the complex workflow underpinning conference management in a more natural way: indeed, agents are viewed as responsible for pursuing activities such as, for instance, retrieving paper information (keywords, title, state of evaluation, . . .), reviewers' information (their research fields, the list of assigned papers, . . .), selecting the possible reviewers for each paper, etc. They can do so by playing roles (possibly evolving over time), such as authors, reviewers, PC-members, etc., and exploiting resources of different granularity—papers, database managers, etc.

In addition, agents can collaborate at a very fine-grained level: in particular, a group of strictly-interacting agents can sometimes be seen as a whole accomplishing a set of goals/tasks. In such a case, we say then that they play a *social role*—a complex role that cannot be played by a single agent because of the many different capabilities (and system resources) required. In the CMS context, a typical social role is the PC-chair, due to the several different activities to be carried out. Some such activities, like the scheduling of the deadlines and the publication of the call for papers, are relatively simple, but others are more complex and therefore call for different entities to work together. This is the case of paper assignment and collecting reviews: in particular, paper assignment requires autonomous “intelligent” entities, able to perform articulated interactions, while respecting key constraints like the familiarity of each reviewer with paper subjects, authors not reviewing their own papers, etc. So, the PC-chair role could be decomposed into a set of “basic roles”, each assigned to a suitable agent. The result is a very flexible system,

which can easily adapt to changes in the system organisation or in the system constraints, with little modifications at the implementation level.

Moreover, the above MAS approach should also count on adequately-expressive forms of (inter-)agent interaction in a distributed scenario—which, in turn, call for suitable design metaphors and run-time entities usable as *coordination media*; altogether, these should be able to enforce the correctness and integrity rules that guarantee the key system properties. For this approach to be effective, the corresponding software engineering methodology and process should be explicitly rooted into the agent paradigm – not just an adaptation of general software engineering approaches for component-based or object-oriented systems – so as to exploit the agent abstractions from the earliest requirements analysis phase, and considering the issue of modelling the environment as one of the most fundamental system aspects: that is what Agent-Oriented Software Engineering is for.

CMS & Agent-Oriented Software Engineering The paradigms and abstractions used in traditional software engineering are inadequate to effectively face the above aspects, as they cannot support the above processes in natural way – in particular, the specification and enactment of coordination patterns. Objects, for instance, are basically neither autonomous nor proactive: their internal activity can be stimulated only by service requests coming from an external thread of control. Moreover, traditional object-oriented applications usually do not have any explicit model of the external “environment”: everything is modelled in terms of objects, which perceive the world only in terms of other objects’ names/references [230] and wrap the environment resources in terms of their internal attributes.

AOSE aims precisely at identifying suitable analysis & design methodologies where the concepts of the agent paradigm (agents, roles, resources, organisational structures) can be fruitfully exploited as the basic building abstractions: each specific methodology then proposes its own view of modelling the system, both in terms of the set of models to define, and of the phases in which the engineering process is articulated. Moreover, the agent-oriented metaphors used to express the system design and architecture should later find their counterpart in suitable MAS items at implementation time. In particular, an AOSE approach typically starts by defining the key roles to be played by agents, and identifying (and suitably expressing) the constraints about inter-role playing that achieve the desired properties: the MAS infrastructure is then asked to guarantee that agents are not allowed to play incompatible roles at any time.

For instance, in conference management, two obvious roles are paper author and paper reviewer: a clear inter-role constraint is that such roles cannot be played simultaneously by the same agent, which means that we must be able to control role assignment and interchange at run time, too. PC-members are another interesting example: in general, they should be authorised to access all the paper reviews in the system, but there should be an exception for their own papers, for obvious fairness reasons. Moreover, some information could be reserved, and rated as accessible only by some predefined roles. To face all such cases, the MAS has to include the capability to check agent operations dy-

namically, so as to possibly prevent agents from accessing reserved information (or part of the system) if needed: in turn, this means that suitable protection should be in place. Of course, unknown external (or unauthorised) agents should be kept out of the system, too—which means that authentication/authorisation schemes should be included.

The above considerations lead to specifying other features that an AO methodology should possess in order to properly support the CMS design:

- *A support for the organisational structure* — Since the organisational structure represents a variable design dimension, the methodology should operate in a *design for change* perspective: the analysis should describe the system requirements without committing to a specific organisational structure, and the architectural design should support several possible architectures. In this way, designers can produce a new detailed design if the conference requirements change. In addition, the methodology should enable the specification of organisational rules that are invariant with respect to any architectural solution.
- *Support for interaction* — Interaction and coordination protocol are clear key issues in CMS: accordingly, the methodology should consider interaction as a key design dimension, making it possible to easily express all the constraints that enable and bound the system interaction space.
- *Support for the security constraints* — Access control is a key problem in CMS, and involves both confidentiality and integrity issues, especially if PC-members are allowed to submit papers. So, the methodology should support some form of access control strategy, like, for instance, Role Base Access Control (RBAC) [175], where the access to a particular resource is granted or denied according to the role currently played by the requesting agent.

CMS & A&A More generally, artifacts make it easier to enrich the MAS design with social and organisational structures, as well as with complex security models: roles, permissions, policies, commitments, and the like can be represented explicitly as first-class entities, and encapsulated within suitable artifacts.

In the context of conference management, artifacts can be exploited to structure and create the *workspaces* used by PC-members to work together and organise the event. For instance, fine-grained artifacts could represent the papers, while larger-grained artifacts could represent the database managers of the system, etc.; of course, choosing one or another kind of artifact is a design dimension which is up to the MAS engineers. Whatever the choice, (intelligent) agents will then use such artifacts to retrieve both the required paper information (keywords, title, state of evaluation, . . .) and the reviewers' information (their research fields, the list of assigned papers, . . .) needed to assign papers to reviewers while respecting all the organisational constraints (such as the maximum number of papers for each reviewer, the minimum number of reviews for each paper, . . .).

Artifacts can also contribute to the system security and integrity, at three levels: by controlling the roles played by agents and their interchange, by denying access to (external) unknown/unauthorised agents, as well as by preventing (known) agents to access some specific, reserved information (or part of the system).

17.4 Conference Management in SODA

This section presents a sketch of a conference management system designed with **SODA**: for obvious space reasons, the core layer has been chosen at a high abstraction level and only a limited set of tables is reported; readers interested in further details can see the Appendix A containing all the tables of the core layer.

The section is organised as follows: Subsection 17.4.1 presents the Requirements Analysis step, Subsection 17.4.2 presents how to move from Requirements Analysis to Analysis, Subsection 17.4.3 shows the Analysis step, while Subsection 17.4.4 explains how to move from Analysis to Architectural Design. Subsection 17.4.5 presents the Architectural Design step, Subsection 17.4.6 shows how to move from Architectural Design to Detail Design by means of the carving operation, and Subsection 17.4.7 presents the Detailed Design step.

17.4.1 Requirements Analysis

The Requirement Analysis step consists of three sets of tables: Requirements Tables, Domain Tables and Relations Tables.

Requirements Tables define the abstract entities tied to the concept of “requirement”: Figure 17.1 illustrates the requirements associated to the actor “Conference Organisers”, which is the only actor individuated in the system at this level of abstraction. In this case, the requirements are related to the “macro activities” composing the conference management workflow already explained and analysed in Section 17.1: startup process, submission process, paper partitioning process, paper assignment process and review process. These are obviously functional requirements – i.e., statements or services the system should provide – but it is also possible to express non-functional requirements—constraints on the services or functions offered by the systems, examples of non-functional requirements are those tied to the system security.

If needed, each requirement could be decomposed into other, more detailed requirements: for instance, the ManagePartitioning requirement could be in-zoomed into a set of requirements about the management of sub-committees, the classification of papers, and paper partitioning. Figure 17.2 shows the corresponding Zooming table $((C)Z_t)$, which formalises the in-zoom of core layer C into the more detailed layer C+1; of course, the same table can be used to represent the dual out-zoom process.

Domain Tables define the abstract entities tied to the concept of “external environment”. In our case, there is only one legacy system, called “WebServer” (Figure 17.3),

Requirement	Description
ManageStartUp	creating call for papers and defining the rules of the organisation
ManageSubmission	managing user registration and paper submissions
ManagePartitioning	partitioning papers based on the conference structure
ManageAssignment	managing the assignment process according to the organisation rules
ManageReview	managing the review process and sending reviews to authors

Figure 17.1: Requirement table $(C)Re_t$

Layer C	Layer C+1
ManagePartitioning	UpdateStartUp ManageSubCommittee ManageClassification PartitionPapers

Figure 17.2: Zooming table $((C)Z_t)$: Paper Partitioning in-zoom

which represents the container for the web application of the conference: the reason to include it in the description is that the conference management system will obviously have to interact with it and these interactions must be captured and constrained.

Legacy-System	Description
WebServer	it is the container for the web application of the conference

Figure 17.3: LegacySystem table $(C)LS_t$

Relations Tables link the abstract entities with each other. In our system, there is just one relation, called “Web”, which involves all the abstract entities, since all requirements need to access the web server to retrieve or store information. The Relation table is illustrated in Figure 17.4.

In Figure 17.5 the Relation-Requirement table at layer C+1 is reported. Since the ManagePartitioning requirement is involved in the Web relation, this relation is “inherited” by all the sub-requirements at layer C+1. In order to maintain consistency of the layer C+1 both the Web relation and the legacy-system WebServer are projected at layer C+1.

Figure 17.6 summarises the outcome of this step. In particular the figure depicts the

Relation	Description
Web	access to the web in order to retrieve or storage some information

Figure 17.4: Relation table $(C)Rel_t$

Requirement	Relation
UpdateStartUp	+ Web
ManageSubCommittee	+ Web
ManageClassification	+ Web
PartitionPapers	+ Web

Figure 17.5: Relation-Requirement table $(C + 1)RR_t$

requirements, the only legacy-system and the relation individuated among requirements and legacy-system (the cyan curved lines) that compose the core layer. In addition the figure shows the two layers that compose the system with the in-zoom operation (the red curved lines) and the projection operation (the black lines).

17.4.2 From Requirements Analysis to Analysis

In order to move from Requirements Analysis to Analysis, the relations between the different abstractions adopted in the two steps must be precisely identified: this is done by means of the References Tables. In particular, the Reference Requirement-Task table $((L)RRT_t)$ specifies the mapping between each requirement and the generated tasks (Figure 17.7).

Requirement	Task
ManageStartUp	start up
ManageSubmission	submission
ManagePartitioning	paper partitioning
ManageAssignment	assignment papers
ManageReview	review process

Figure 17.7: Reference Requirement-Task table $(C)RRT_t$

Typically, relations are not one-to-one, since a requirement is likely to generate several tasks; however, in this case each requirement can be actually mapped into one task, as a consequence of setting the core layer at very high abstraction level. The same consideration applies to the other Reference Tables.

Since our system is composed of two layers, a similar set of tables will be defined also for the C+1 layer. Figure 17.8 shows the Reference Requirement-Task table at layer C+1.

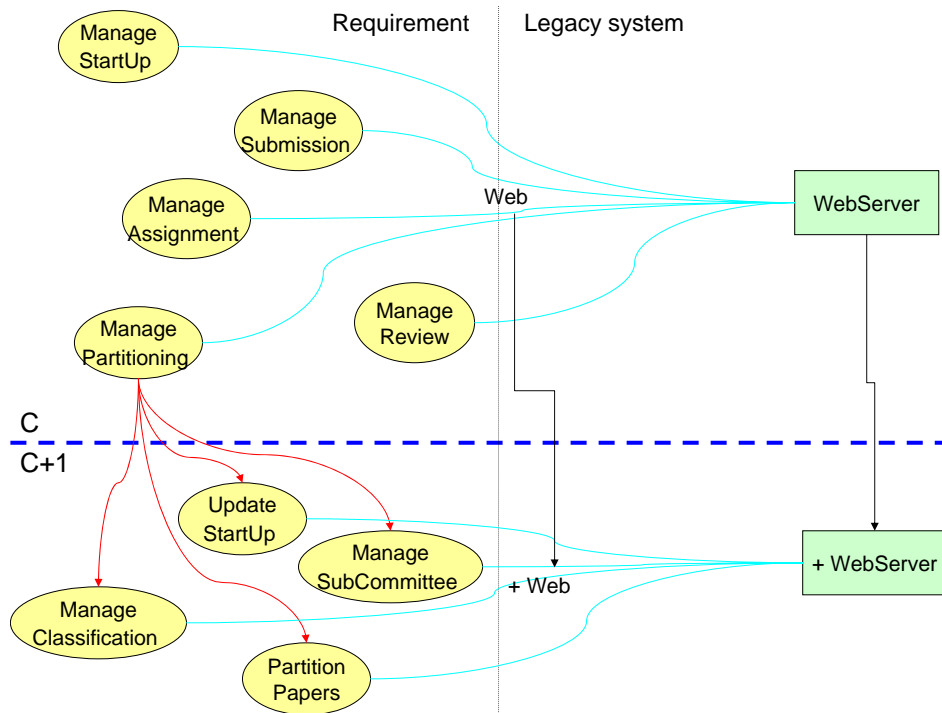


Figure 17.6: An overview of the Requirement Analysis step

Requirement	Task
UpdateStartUp	modifying startup
ManageSubcommittee	create sub-committees Vice-Chair elections
ManageClassification	papers classification
PartitionPapers	partition papers

Figure 17.8: Reference Requirement-Task table $(C + 1)RRT_t$

17.4.3 Analysis

The Analysis step exploits three sets of tables: Responsibilities Tables, Dependencies Tables and Topologies Tables.

Responsibilities Tables define the abstract entities tied to the concept of “responsibilities centre” – namely, tasks and functions. The conference management tasks have been already reported in Figure 17.7, while the functions are reported in Figure 17.9. The last function in the table is the only one coming from the legacy system: all the others are generated from the requirements in order to support the achievement of tasks.

Function	Description
management user	managing user information
management review	managing review information
management paper	managing paper information
management assignment	managing assignment information
management partitioning	managing partitioning information
management process	managing start-up information
webSite	web interface of the conference

Figure 17.9: Function table $(C)F_t$

In Figure 17.10 the Zooming Table for the Analysis step is reported. In this table both the sub-tasks of the paper partitioning task and the new dependencies – underlined in the table – generated by the in-zoom operation are listed. These dependencies coordinate the execution of sub-tasks at layer C+1 in order to achieve the original task at layer C.

Layer C	Layer C+1
paper partitioning	modifying startup create sub-Committees Vice-Chair elections papers classification partition papers <u>new organisation</u> <u>classification</u> <u>partition</u> <u>election</u>

Figure 17.10: Zooming table $(C)Z_t$

Dependencies Tables relate functions and tasks with each other. Figure 17.11 shows the dependencies for conference management: “webAccess” derives from the relation defined in the Requirement Analysis, while the others come from the relationship among tasks and functions.

Dependency	Description
start up information	access of all the information about start up process
user information	access to all the users' information
paper information	access to all the paper information
partitioning information	access to all the information about partitioning process
submission information	access to all the information about submission process
assignment information	access to all the information about assignment process. A reviewer cannot be the author of the papers that are assigned to him
review information	access to all the information about review process
webAccess	access to the web site of the conference

Figure 17.11: Dependency table $(C)D_t$

Topologies Tables, in turn, express the topological constraints over the environment. As a web-based system, the conference management system does not impose many topological constraints over the environment: in fact, one constraint is identified—the locus where the functions are allocated.

17.4.4 From Analysis to Architectural Design

In order to link the Analysis step with the Architectural Design step, the Analysis entities are related to the Architectural Design by means of Transition Tables. As an example, Figure 14.13 shows the mapping between tasks and roles: in this case, three different tasks are assigned to the same role (PC-chair) because the capabilities requested for the achievement of the three tasks are similar. Since our system is composed of two layers, a similar set of tables will be defined also for the C+1 layer.

Role	Task
PC-chair	paper partitioning start-up assignment papers
Author	submission
PC-member	review process

Figure 17.12: Transition Role-Task table $(C)TRT_t$

17.4.5 Architectural Design

The Architectural Design step consists of three sets of tables: Entities Tables, Interaction Tables and Topological Tables.

Entities Tables describe both the active entities (the roles) able to perform some action in the system, and the passive entities (the resources) which provide services. Figure 17.13 reports the Role-Action table for conference management.

Role	Action
PC-chair	login, assignment publish deadline partition
Author	login, send paper
PC-member	login, read paper write review download paper

Figure 17.13: Role-Action table $(C)RA_t$

Interaction Tables describe the interaction between roles and resources: Figure 17.14 shows only a subset of the rules for conference management—namely, the organisational rules. For example, the AutRev-Rule specifies that an author cannot be the reviewer of his/her own paper, while the Author-Rule specifies that the author can access only the public information about his/her own paper even if he/she is a PC-member.

Interaction	Description
Deadline-Rule	send paper is possible if and only if time is minus then deadline submission
User-Rule	get user is possible if the request user is the requester or the requester is the PC-chair
Author-Rule	author can access and modify only his public paper information
Match-Rule	papers can be partitioned according key words
AutRev-Rule	the PC-member cannot be the author of paper
Review-Rule	the PC-member cannot access to private information about his papers
Access-Rule	the access to the system must be authorised

Figure 17.14: Interaction table $(C)I_t$

Finally, Topological Tables describe the logical structure of the environment. As outlined above, the conference management system does not present a complex topology: so, a single workspace is enough for representing the logical structure of the environment.

17.4.6 From Architectural Design to Detailed Design

Now, in order to transit to the Detailed Design phase, the “carving operation” has to be performed. Figure 17.15 - a) shows the key layers of our system. Since the three roles at layer C derive from the same role (“Organisation”) at layer C-1, the carving operation is very simple: the three roles at layer C become agents, and the role at layer C-1 becomes a society (Figure 17.15 - b). The result is the set of Mapping Tables, which relate the Architectural Design entities to the Detailed Design.

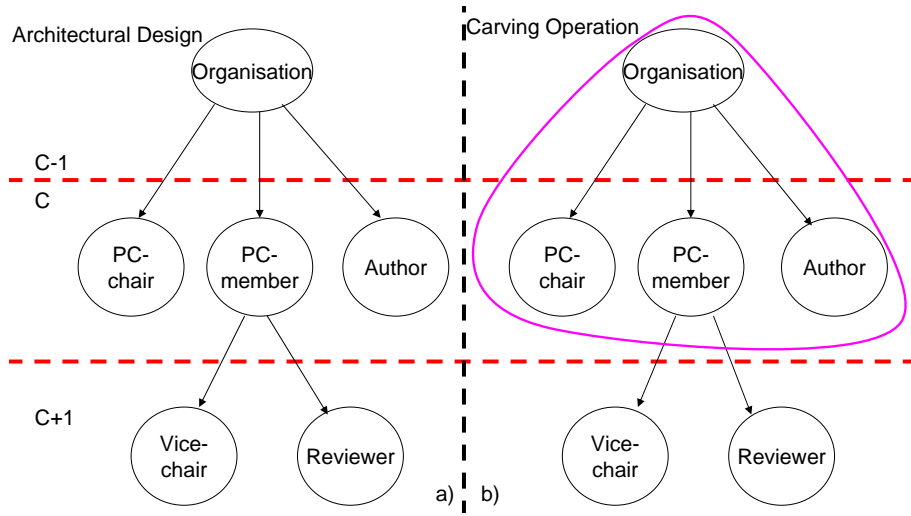


Figure 17.15: Carving operation in the Conference Management

In conference management, each role is assigned to a specific agent (Figure 17.16) and each resource is assigned to a specific (environmental) artifact; on the contrary, some interactions could be mapped onto the same (social) artifact, as in the case of the Author-Rule and the Match-Rule, because both rules are used in the partitioning process and they are enforced in the same society.

Agent	Role
PC-Chair Agent	PC-Chair
Author Agent	Author
PC-Member Agent	PC-member

Figure 17.16: Mapping Agent-Role table $(C)MAR_t$

17.4.7 Detailed Design

The Detailed Design step exploits two sets of tables: Agent/Society Design Tables, and Environment Design Tables. The first set of tables depicts agents, individual artifacts, and the agent societies derived from the carving operation.

Figure 17.17 reports the Society-Artifact table for conference management: as discussed above, there is only one agent society, along with the social artifacts used for enforcing the corresponding social rules.

Society	Artifact
Org	StartUp Artifact User-Rule Artifact Partitioning Artifact Assignment Artifact Review Artifact

Figure 17.17: Society-Artifact table $(C)SAr_t$

In their turn, Environment Design Tables concern the design of artifacts and workspaces: Figure 17.18 reports an excerpt of the Artifact-UsageInterface table: as an example, the Review Artifact features two operations, each implying a check of the user's credentials.

Artifact	Usage Interface
StartUp Artifact	deadline extension, update rule, read rule
User-Rule Artifact	get user, modify user
Partition Artifact	partition paper, access classification
Assignment Artifact	check authors, check reviewers
Review Artifact	access review information, insert review

Figure 17.18: Artifact-UsageInterface table $(C)AUI_t$

The system design at layer C is now complete. Of course, this is still quite a high-level system view: several in-zoom operations are needed, until the system design is refined enough for implementation purposes.

17.5 From the design to a TuCSoN-based implementation

As highlighted in Chapter 16 the SODA design phase could be mapped onto AO infrastructures. In order to complete the case study a mapping between SODA design and the TuCSoN infrastructure is presented. In Figure 17.19 an excerpt of the figure already showed in Chapter 16 is reported. This figure reports only the columns related to SODA and TuCSoN.

SODA	TuCSoN
<i>Role</i>	<i>Role</i>
<i>Action</i>	<i>Coordination Primitive</i>
<i>Interaction</i>	<i>Reaction Specification</i> <i>Reaction</i>
<i>Operation</i>	<i>Tuple Centre Operation</i>
<i>(Social) Artifact</i>	<i>Tuple Centre</i>
<i>(Individual) Artifact</i>	<i>ACC</i>
<i>(Environmental) Artifact</i>	-
<i>Aggregate</i>	<i>Linked Tuple Centres</i>
<i>Society</i>	<i>Organisation</i>
<i>Workspace</i>	<i>Node</i>
<i>Workspace Connection</i>	-

Figure 17.19: Abstractions Mapping

The three roles individuated in the Architectural Design step – PC-chair, Author and PC-member – are mapped to the corresponding three **TuCSoN** roles and as a consequence these three roles are played by three different agents (Figure 17.16). In **SODA** each agent is associated to an (individual) artifact (PC-Chair Artifact, Author Artifact and PC-Member Artifact) – see Appendix A for the complete case study – which generates the creation of three different ACCs in **TuCSoN**, each of them associated to a specific **TuCSoN** agent.

The actions associated to the roles and executed by agents are transformed to coordination primitives. For example the action “write review” (see Figure 17.13) is mapped onto the coordination primitive [166] “out”.

The operation associated to the resources and performed by artifacts are transformed to the tuple centre operations. For example the operation “store review” that supports the action “write review” is mapped onto:

```
PaperArt?out(review(pc-member, paper_id, rev))
```

where “PaperArt” is the name of the tuple centre that stores the information about papers and “review” is the tuple posted in the tuple centre. The tuple’s arguments are the id of the PC-member that has done the review (pc-member), the id of the paper (paper_id) and finally the review made by PC-member (rev) [146].

The interactions individuated in the Architectural Design step (Figure 17.14) are transformed in reactions. For example let us suppose that a PC-member needs to download a specific paper for making the review. This action is critical from the fairness point of view because a PC-member could be the author of that paper; in fact the action is subject to the “Review-Rule”. So the PC-member should invoke the operation

```
PaperArt?in(download(paper_id, Paper_link))
```

onto “PaperArt” tuple centre. The tuple centre should execute the reaction

```
reaction( in(download(Paper_id, Paper_link)), (request, from_agent),
(
  current_source(Agent),
  rd(association(Agent, PC_member)),
  rd(authorised(PC_member, Paper_id)),
  rd(link(Paper_id, Link)),
  out(download(Paper_id, Link))
)).
```

```
reaction( in(download(Paper_id, Paper_link)), (request, from_agent),
(
  current_source(Agent),
  rd(association(Agent, PC_member)),
  no(authorised(PC_member, Paper_id)),
  out(download(Paper_id, nil))
)).
```

If the PC-member is authorised to download the paper the tuple retrieved will contain the link to the paper, otherwise the tuple will contain a null pointer.

As highlighted in Subsection 17.4.6, in conference management a society (Org) created by the carving operation is presented, so this society is mapped onto an organisation in **TuCSon**. This organisation will be responsible for the creation and management of the ACCs representing the individual artifacts. The environmental artifacts are not supported by **TuCSon** so these abstractions must be created in an ad hoc way by developers.

Finally the **SODA** design presents only one workspace so a single **TuCSon** node is enough for this system.

17.6 Discussion

The requirements of our agent-based CMS derive partially from CMS general issues (Section 17.2), and partially from its concretisation within the agent-oriented approach (Section 17.3): altogether, they determine the features required by an agent-oriented methodology to be suitably used for CMS analysis and design. Accordingly, it is first discussed to which extent **SODA** meets the general CMS issues, then the discussion is extended to the latter requirements—in both cases, comparing the **SODA** approach with some other major methodologies (Gaia [229, 228] and O-MaSE [43, 41]), which considered the same CMS case study. In addition this case study was already designed with the **SODA** early

version – called **SODA-EV** in the reminder of this section – by Emanuela Mattiolo in her Masters thesis [118]. The Mattiolo’s thesis was written in italian so here it is reported only a qualitative explanation of the work.

Let us consider first the four basic requirements highlighted in Section 17.2:

1. *“The whole set of actors and interaction should be modelled using suitable high-level metaphors . . .”* In **SODA** the whole set of actors, resources, and interactions of the CMS has been modelled using the (high-level) metaphors of the A&A meta-model – both becoming actual “live” entities at run-time. In the two other methodologies this issue is only partially addressed: both Gaia and O-MaSE model agents and interaction, but the environment is not fully captured. Indeed, Gaia provides support for the modelling of the environment only in the analysis and architectural design phase, while O-MaSE does not provide any support for this issue. Finally in **SODA-EV** the whole set of actors, resources, and interactions has been modelled using agents, coordination media and the relatives coordination rules, and infrastructure classes. The outcome is similar to that obtained from **SODA** however the abstractions used by **SODA-EV** are not so expressive like those used in the new version of the methodologies.
2. *“It should be possible to specify any kind of coordination pattern (...) to coordinate several activities.”* The coordination and interaction problems are addressed by **SODA** artifacts, which on the one side model the MAS environment, on the other enable/constrain the interactions among agents. Again, Gaia and O-MaSE address this issue only partially, as there is no way to express complex coordination patterns—both support only one-to-one interaction protocols. Indeed, this aspects in **SODA-EV** is well addressed by means of coordination media and coordination rules. However these abstractions can be exploited only in the context of the agents societies, coordination medium cannot be used alone.
3. *“Adequate organisational abstractions should be available to express organisational constraints, environmental aspects and security issues (...)”* The organisational structure is modelled via **SODA** society abstraction, along with its social artifacts. In particular, some social artifacts are exploited to enforce both the organisational rules (the definition of the deadline, the structure of the event,etc.) and the security policies for access control and confidentiality. Both Gaia and O-MaSE support the design of the organisational structure, but only Gaia allow the designer to express organisational constraints: in particular, environmental aspects and security issues are not addressed. In **SODA-EV** the organisational structure is modelled via societies of agent that exploits coordination medium in order to coordinate the work of agents belonging to the society. The coordination medium enforces the organisational rules but it is not able to enforce the security policy for the access control, so if complex and fine-grained policies are necessary a specific infrastructure class should be designed for this purpose.

4. “*The software infrastructure should be based on open architectures, so as to support dynamic evolution (...)*”. The output of the **SODA** design phase is easily supported by several different infrastructures (Chapter 16) – for instance, **TuCSon** (see Section 17.5) , **TOTA** and **CARTAGO** - which provide native support for coordination among agents. Gaia and O-MaSE do not directly deal with implementation issues. The outcome of the two processes is a technology-neutral specification, that the developer needs to “adapt” to the specific, selected technology. The outcome of the **SODA-EV** design is supported by **TuCSon**: coordination medium and coordination rules are respectively mapped in tuple centre and reactions while infrastructure classes should be designed in an ad hoc way because they are not supported by **TuCSon**.

The other features required from an AO methodology to properly support the CMS design (Section 17.3) concern three main aspects: the support for the organisational structure, the support for interaction, and the support for the security constraints. With respect to such issues, the **SODA** approach is compared to Gaia O-MaSE and **SODA-EV** .

Support for the organisational structure — In **SODA**, the layering principle supports scalability by allowing different system architectures to be managed in a uniform way: so, any change in the organisational structure implies – in principle – only a different carving and some little modifications in the Detailed Design.

For instance, if a (small) conference becomes larger, the basic Program Committee structure, made up of a PC-chair and PC-members, may no longer be adequate for the purpose: there might be need for sub-committees, each with its own Vice-chair. To take care of this change, it is necessary to go back to the Architectural Design step and adopt a different carving, which, in turn, calls for a revision in the Detailed Design. However, since the Vice-chair in the sub-committee plays the same role as the PC-chair in the Program Committee, the basic conference structure can be reused in the sub-committees, which are governed by the same interaction rules defined for the basic conference structure; correspondingly, the related social artifacts can be reused with no changes. Analogously, the agent individual artifacts of the basic conference structure can be reused for the agents belonging to sub-committees, with minimal changes – just those required by the 1-1 relationship holding between each agent and its individual artifact.

Instead, Gaia, O-MaSE and **SODA-EV** do not provide any similar mechanism for managing complexity. In particular, Gaia requires that the (assumedly better) organisational structure is chosen in the architectural design phase, and it does not support the creation of different architectural designs: so, if, after that step, the organisation structure changes for any reason – like the introduction of sub-committees as above – the designer is forced to redo the system design from scratch. Moreover, only guidelines are provided to support the designer in such a key choice. Similar considerations hold for both O-MaSE and **SODA-EV** , too: specific CMS organisational problems are simply not considered. So, once again, if the organisational structure changes the designer is forced to restart the

development cycle from the first analysis phase.

Furthermore, neither Gaia nor O-MaSE provide any abstraction to encapsulate and enforce the organisational rules, like **SODA** social artifacts: so, agents are left alone in dealing with the enforcing of such rules by themselves. Indeed, **SODA-EV** provides the coordination medium abstraction that enforces the organisational rules.

Support for interaction — As highlighted above, this issue is addressed in **SODA** by means of suitable artifacts: in particular, a social artifact enables and bounds the agent behaviour by applying the organisational rules and the coordination laws underpinning the CMS workflow, while an individual artifact provides the agent with all the “sensors and actuators” needed for living in a MAS. Accordingly, it stores the protocols for each role played by an agent and contains the list of all the actions that an agent can perform when playing a given role.

In **SODA-EV** interactions are modelled in terms of interaction protocols and interaction rules. Interaction protocols do not provide support for expressing the actions that agents can perform so they are not so suitable for modelling the interaction of cognitive agents. In a similar way the interaction rules support coordination in the agent societies but they are not able to express constraints over the agents’ interaction space.

In the other two methodologies, instead, the interaction and coordination protocol issues are only partially addressed. More precisely, the Gaia interaction model is based on one-to-one interaction protocols, which seem inadequate and not expressive enough, for supporting the complex coordination patterns required by CMS: for instance, no constraints over the interaction space can be easily expressed, nor are there any specific abstractions to properly enforce the coordination laws as encapsulated in social artifacts. Similar consideration can be done for O-MaSE, too. Due to these limitations, Gaia and O-MaSE may lead to systems that are not flexible enough: in particular, each time the coordination patterns change, the designer is forced to re-design both all the entities and the interaction protocols involved, because the coordination is *spread* all over these entities and protocols. This is not the case in **SODA**, where the designer should just re-design the specification of the modified coordination laws.

Support for the security constraints — Security policies such as Role-Based Access Control (RBAC) [175, 177] can be easily designed in **SODA**, by just associating a social artifact to each environmental artifact so as to “implement” the RBAC policy; as a result, each time an agent needs to access an environmental artifact, the social artifact can grant access based on the role currently played by the agent.

In Gaia, O-MaSE and **SODA-EV**, instead, the security issue is only partially addressed. In Gaia, for instance, some access policies can be expressed by means of the organisational rules: however, no specific abstraction is provided to encapsulate such rules and enforce them at run time, so agents may have to take care of this issue by themselves. In a similar way, **SODA-EV** could express access policies by means of coordination rules: however this rules was not designed for this purpose so coordination rules can express only simple and coarse-grained access policies. On the other hand, O-MaSE

support for security is quite limited, as there is no way to express access constraints without designing a “special role / agent” for each resource that checks the accessed operation – a very expensive approach in terms of computational resources.

Summing up, **SODA** seems better equipped than other methodologies – and than its previous version **SODA-EV** – when facing many CMS issues, including the ability to operate in a *design-for-change* perspective. Indeed, **SODA** is able to meet all the above features at a satisfactory level, proving to be an effective tool for the CMS case study: in particular, its orientation towards “inter-agent” issues, like the engineering of agent societies and the interactions between agents and environment, helps to model the CMS at an adequately-natural abstraction level, with a direct counterpart at the implementation level in infrastructures.

Part VI
Conclusion

18

Conclusion and Research Directions

This chapter presents a summary of this thesis. In particular, Section 18.1 summarises the contributions of the thesis and Section 18.2 presents the plans for the future works.

18.1 Summary of the Contributions

This thesis can be considered as a bidirectional path from concepts at meta-model level down to concepts at infrastructure level and back, centered on agent-oriented methodologies for the engineering of complex software systems. Several different sources have driven the main contribution of this thesis: the extension of the **SODA** methodology (Chapters 14, 15), and its adoption for the design, development and management of some classes of software systems requiring articulated form of interactions and support for environment engineering, such as conference management systems (Chapter 17). In particular:

- *Methodologies* — my contribution in this thesis is a detailed presentation and comparison of the state of art of AO methodologies (Chapters 3, 4 and 6)
- *Meta-models* — my contributions in this thesis are:
 - a detailed presentation of the state of art of meta-model techniques and a review of different meta-modelling languages proposed in the literature (Chapter 5)
 - a proposal of a technique for meta-modelling the infrastructures. As for methodologies where two different types of meta-model are proposed – one for abstractions and the other for the process – the technique that I have proposed is for meta-modelling both the static part of an infrastructure – i.e., the run-time abstractions and their relationships supported by the infrastructure – and the dynamic part—the run-time abstractions behaviour (Chapter 5).
 - a proposal of the adoption of the A&A meta-model for modelling and designing the agent-oriented systems. The agent abstraction sometimes seems too complex for modelling some parts of the MAS like for example the environment where agents live and interact. The environment's resources typically have

a passive, function-oriented behaviour that is not properly addressed by the agents that are pro-active and autonomous entities. So the agent abstraction is not enough for designing MASs, a new function-oriented abstraction should be introduced for design the MAS environment. The abstraction proposed in this thesis for design the environment is the artifact (Chapter 7).

- *Environment* — my contributions in this thesis are:
 - classifying AO methodologies along the dimension of environment support, and grouping them in three different categories: (*strong-env*) strong environment viewpoint—methodologies that support both modelling and design of MAS environment; (*weak-env*) weak environment viewpoint—methodologies that support only the modelling of MAS environment; (*no-env*) no environment viewpoint—methodologies that do not explicitly model MAS environment (Chapter 8)
 - a proposal of a mechanism for introducing the environment in those methodologies that not support or support only partially the environment. Starting from no-env AO methodologies, I suggest how to transform them in weak-env methodologies, and subsequently in strong-env methodologies.
 - a detailed presentation of the state of art of of the agent-oriented infrastructures and the development of the meta-models of some agent-oriented infrastructures (Chapter 10).
- *Complexity management* — my contributions in this thesis are:
 - a presentation of the state of the art of the complexity management both in object-oriented and in agent-oriented methodologies (Chapters 11 and 12).
 - a proposal of the *layering principle* in order to manage the complexity in the systems representations (Chapter 12).

These studies have led to the formulation of a new version of **SODA** addressing the limitations of the original version (Chapters 14 and 15). **SODA** now is the first agent-oriented methodology that explicitly adopted mechanisms for scaling the complexity of the system representations by means of the layering principle. The new version of **SODA** supports environment engineering by adopting artifacts as the second milestone for MAS modelling and engineering, side-by-side to agents, clearly distinguishing between agents and the entities they use—i.e., between goal/ task-driven entities, and entities whose goal is assigned by agents at the time of their usage. The modelling and design of the **SODA** core element – i.e., interaction space – are improved by the adoption of concepts such as “action” – that agents are able to perform – “operation” – that artifacts are able to provide – and “interaction”—that enables and bounds the interaction space. In addition now **SODA** has well-defined and formally described meta-models both for the abstractions

and for the process underpinning the methodology. Finally in order to fill the gap between methodologies and infrastructure we have proposed a general method based on the meta-models mapping (Chapter 16): the concepts of the **SODA** design phase are mapped to the concepts of three different AO infrastructures—**TuCSon**, **CARtAgO** and **TOTA**.

18.2 Research Directions

The thesis has not exhausted the space of investigations opened with the adoption of A&A meta-model and layering principle. Among the research directions which we consider most interesting and worth of exploration as a natural next step following this thesis, we have:

Tools & Languages — The tabular representation adopted by **SODA** is powerful because it allows to model and design in a clear and ordered way all the entities and their relative features, relationships and constraints. However this kind of representation seems more suitable for an automatic tool than for a human one. In fact, currently it is not so easy to model a system with **SODA** because there is not an automatic tool that support the designer. So, one of the first new contributions will be the development of a tool for supporting the designers. In addition we are planning to develop a graphical language for modelling and designing systems with **SODA**. This new language will not replace the tabular representation but it will provide a different view of the system like in OPM which adopts both a graphical and textual description.

Another interesting extension of **SODA** could be involved in the creation – or adoption – of a further language for specifying in a more suitable way the **SODA** interactions. At the moment the natural language supported by the tabular representation seems to be ambiguous and not so suited for expressing rules over the agents’ interaction space.

Fragmentation & Intra-agent issue — The development of complex software systems using the agent-oriented approach requires suitable methodologies which provide explicit support for the key abstractions of the agent paradigm [36]. To date, several methodologies supporting the analysis, design and implementation of MAS have been proposed in the context of AOSE (Chapter 4). Although such methodologies have different advantages when applied to specific problems, it is a fact that a unique methodology cannot be general enough to be useful for everyone without some level of customisation. Thus an approach that combines the designer’s need to define his/her own methodology with the advantages and the experiences coming from the existing and documented methodologies is required. A possible solution to this problem is to adopt the *method engineering paradigm* [15, 16], thus enabling designers of MAS to (re)use parts coming from different methodologies in order to build up a customised approach to their own problems. According to this approach, the “development methodology” is constructed by assembling pieces of other methodologies (*method fragments*) from a repository of methods (*method base*).

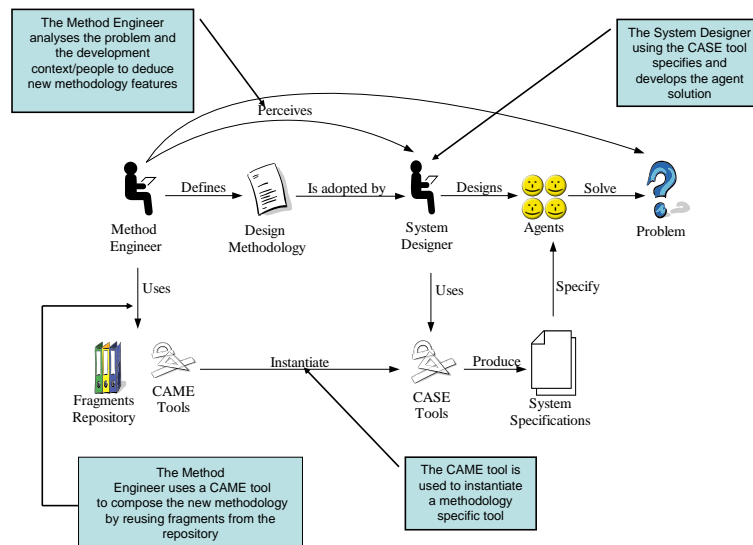


Figure 18.1: FIPA Method Engineering Idea [121]

This approach has been adopted, in the past few years, by the FIPA Methodology Technical Committee (TC) (FIPA Foundation for Intelligent Physical Agents)[121](Figure 18.1).

Accordingly, our work could follow two different paths:

- on one hand the adoption of one or more fragments dealing with the internal agent architecture can address the intra-agent issue which is still not considered in **SODA**; for example this new fragment could be integrated in **SODA** by exploiting the layering principle and specialising the zooming mechanism. In-zoom could have two different employments: (i) the current “explosion” of agents into societies and (ii) the design of the internal agent architecture.
- on the other hand, the extraction of different fragments from **SODA** could be useful for other AO methodologies. For example fragments like ones dealing with environment engineering or with the layering principle could be easily adopted by the no-env or weak-env methodologies (Chapter 8).

Artifacts Engineering — As for the intra-agent issue, currently **SODA** does not provide support for the internal engineering of artifacts. Following the above ideas it will be possible to specialise also the zooming for the artifacts. How to design the internal architecture of artifacts is currently under investigation [180] by our research group rooted in Cesena and at the time of writing there is no definitive solution. As long there is no a definitive solution it is possible to use the traditional object-oriented design for the artifacts.

AOSE & Security — Security is the process of protecting data from unauthorised

access, use, disclosure, destruction, modification, or destruction [120]. These fields are interrelated and share the common goals of protecting the confidentiality, integrity and availability of information; however, there are some subtle differences between them. These differences lie primarily in the approach to the subject, the methodologies used, and the areas of concentration [111]. Information security is concerned with the confidentiality, integrity and availability of data regardless of the form the data may take. In the context of MAS access control seems the key security problem, and it involves both the confidentiality and the integrity issues. The integration of access control techniques – such as RBAC [175] – or security techniques – such as risk analysis – in general inside AO methodologies is in its early stage of investigation [133]. We are planning an extension of **SODA** for addressing the access control issue.

Architectural Styles & AOSE — An architectural style specifies a vocabulary of component and connector types and a set of constraints defining these elements can be combined [196]. Some investigations in the context of MAS and software architecture is done [76] and different architectural styles are already developed [108, 125, 126] but other deep investigations are necessary for developing suitable architectural styles for MAS.

In addition to these research directions, we consider of primary importance to put to test **SODA** with a number of different case studies belonging to different areas ranging from workflow management since to biological applications [119] which are very complex applications that can stress the methodology. These tests will be very useful for the improvement of the methodology.

Part VII
Appendix

A

The Complete Case Study

This Appendix presents all the tables for the conference management system designed with **SODA** not reported in Chapter 17: for obvious space reasons, the core layer has been chosen at a high abstraction level.

A.1 Requirements Analysis

The Requirement Analysis step consists of three sets of tables: Requirements Tables, Domain Tables and Relations Tables.

Requirements Tables define the abstract entities tied to the concept of “requirement”: Figure A.2 illustrates the requirements associated to the actor “Conference Organisers”, which is the only actor individuated in the system as illustrated in Figure A.1. The requirements are related to the “macro activities” composing the conference management workflow: startup process, submission process, paper partitioning process, paper assignment process and review process.

Actor	Requirement
Conference Organisers	ManageStartUp ManageSubmission ManagePartitioning ManageAssignment ManageReview

Figure A.1: Actor-Requirement table $(C)AR_t$

Requirement	Description
ManageStartUp	creating call for papers and defining the rules of the organisation
ManageSubmission	managing user registration and paper submissions
ManagePartitioning	partitioning papers based on the conference structure
ManageAssignment	managing the assignment process according to the organisation rules
ManageReview	managing the review process and sending reviews to authors

Figure A.2: Requirement table $((C)Re_t$

If needed, each requirement could be decomposed into other, more detailed requirements: for instance, the ManagePartitioning requirement could be in-zoomed into a set of requirements about the management of both sub-committees and papers.

Figure A.3 shows the corresponding Zooming table $((C)Z_t)$ for the Requirement Analysis step, which formalises the in-zoom operation of core layer C into the more detailed layer C+1.

Layer C	Layer C+1
ManagePartitioning	UpdateStartUp ManageSubCommittee ManageClassification PartitionPapers

Figure A.3: Zooming table $((C)Z_t)$: Paper Partitioning in-zoom

Figure A.4 shows the Requirement Table at layer C+1, where the four requirements generated by the in-zoom operation are described. In particular the first two requirements (UpdateStartUp and ManageSubCommittee) are devoted to managing the organisation structure of the conference – updating conference rules and/or creating sub-committees – while the last two requirements are devoted to managing both the classification and the partitioning of the submitted papers.

Requirement	Description
UpdateStartUp	It could be necessary to update the structure and the rules of the organisation in order to manage the great number of paper submitted
ManageSubCommittee	If it is necessary the sub-committes will be created
ManageClassification	classification of the papers according to key words suggested by authors
PartitionPapers	partitioning of papers in order to accomplish at the organisation's rules

Figure A.4: Requirement table $(C + 1)Re_t$

Domain Tables define the abstract entities tied to the concept of “external environment”. In our case, there is only one legacy system, called “WebServer”, which represents the container for the web application of the conference: the reason to include it in the description is that the conference management system will obviously have to interact with it and this interaction must be captured and constrained. Figure A.5 presents the ExternalEnvironment-LegacySystem table and Figure A.6 presents LegacySystem Table where the WebServer system is described.

External-Environment	Legacy-System
External	WebServer

Figure A.5: ExternalEnvironment-LegacySystem table $(C)EELS_t$

Legacy-System	Description
WebServer	it is the container for the web application of the conference

[h]

Figure A.6: LegacySystem table $(C)LS_t$

Relations Tables link the abstract entities with each other. In our system, there is just one relation, called “Web”, which involves all the abstract entities, since all requirements need to access the web server to retrieve or store information. The Relations Tables are illustrated in Figures A.7, A.8 and A.10.

Relation	Description
Web	access to the web in order to retrieve or storage some information

Figure A.7: Relation table $(C)Rel_t$

Requirement	Relation
ManageStartUp	Web
ManageSubmission	Web
ManagePartitioning	Web
ManageAssignment	Web
ManageReview	Web

Figure A.8: Relation-Requirement table $(C)RR_t$

In Figure A.9 the Relation-Requirement table at layer C+1 is reported. Since the ManagePartitioning requirement is involved in the Web relation, this relation is “inherited”, in this case, by all the sub-requirements at layer C+1. In order to maintain the consistency of the layer C+1 both the Web relation and the legacy-system WebServer are projected at layer C+1. The tables related to legacy-system and to relation are not reported in the layer C+1 because are already presented at layer C with the exception of the label “+” indicating the projection.

Requirement	Relation
UpdateStartUp	+ Web
ManageSubCommittee	+ Web
ManageClassification	+ Web
PartitionPapers	+ Web

Figure A.9: Relation-Requirement table $(C + 1)RR_t$

Legacy-System	Relation
WebServer	Web

Figure A.10: Relation LegacySystem table $(C)RLS_t$

A.2 From Requirements Analysis to Analysis

In order to move from Requirements Analysis to Analysis, the relations between the different abstractions adopted in the two steps must be precisely identified: this is done

by means of the References Tables. In particular, the Reference Requirement-Task table ($(L)RRT_t$) specifies the mapping between each requirement and the generated tasks (Figure A.11).

Requirement	Task
ManageStartUp	start up
ManageSubmission	submission
ManagePartitioning	paper partitioning
ManageAssignment	assignment papers
ManageReview	review process

Figure A.11: Reference Requirement-Task table $(C)RRT_t$

Typically, relations are not one-to-one, since a requirement is likely to generate several tasks; however, in this case each requirement can be actually mapped onto one task, as a consequence of setting the core layer at a very high abstraction level. The same consideration applies to the other Reference Tables illustrated in Figures A.12, A.13, A.14 and A.15.

Requirement	Function
ManageStartUp	management process management user
ManageSubmission	management user management paper
ManagePartitioning	management partitioning management paper
ManageAssignment	management assignment management paper
ManageReview	management review management paper

Figure A.12: Reference Requirement-Function table $(C)RRF_t$

Legacy-System	Function
WebServer	webSite

Figure A.13: Reference LegacySystem-Function table $(C)RLSF_t$

Legacy-System	Topology

Figure A.14: Reference LegacySystem-Topology table $(C)RLST_t$

Relation	Dependency
Web	webAccess

Figure A.15: Reference Relation-Dependency table $(C)RRD_t$

Since our system is composed of two layers, a similar set of tables will be defined also for the C+1 layer. Figure A.16 shows the Reference Requirement-Task table at layer C+1.

Requirement	Task
UpdateStartUp	modifying startup
ManageSubcommittee	create sub-committees Vice-Chair elections
ManageClassification	papers classification
PartitionPapers	partition papers

Figure A.16: Reference Requirement-Task table $(C + 1)RRT_t$

A.3 Analysis

The Analysis step exploits three sets of tables: Responsibilities Tables, Dependencies Tables and Topologies Tables.

Responsibilities Tables define the abstract entities tied to the concept of “responsibilities centre” — namely, tasks and functions. The conference management tasks are reported in Figure A.17, while the functions are reported in Figure A.18. The last function in the table is the only one coming from the legacy system: all the others are generated from the requirements in order to support the achievement of tasks.

Task	Description
start up	insertion of the setup information
submission	submission of paper
paper partitioning	partitioning of the set of papers
assignment papers	assignment papers to PC-members
review process	creation and submission of the reviews

Figure A.17: Task table $(C)T_t$

Function	Description
management user	managing user information
management review	managing review information
management paper	managing paper information
management assignment	managing assignment information
management partitioning	managing partitioning information
management process	managing start-up information
webSite	web interface of the conference

Figure A.18: Function table $(C)F_t$

In Figure A.19 the Zooming Table for the Analysis step is reported. In this table the sub-task of the paper partitioning task are listed. In addition the table contains the list of the new dependencies – underlined in the table – generated by the in-zoom operation. The tasks of layer C+1 are then reported in Figure A.20.

Layer C	Layer C+1
paper partitioning	modifying startup create sub-committees Vice-Chair elections papers classification partition papers <u>new organisation</u> <u>classification</u> <u>partition</u> <u>election</u>

Figure A.19: Zooming table $(C)Z_t$

Task	Description
modifying startup	update the structure and the rules of the organisation
create sub-committees	creating of sub-committees
Vice-Chair elections	for each sub-committee it is necessary to elect the Vice-Chair
papers classification	classification of papers in base of key words
partition papers	partitioning papers in base of their classification

Figure A.20: Task table $(C + 1)T_t$

Dependencies Tables relate functions and tasks with each other. Figure A.21 shows the dependencies for conference management: “webAccess” derives from the relation defined in the Requirements Analysis, while the others come from the relationship among tasks and functions. The links among tasks and dependencies are reported in Figure A.22, while the links among functions and dependencies are depicted in Figure A.23.

Dependency	Description
start up information	access of all the information about start up process
user information	access to all the users' information
paper information	access to all the paper information
partitioning information	access to all the information about partitioning process
submission information	access to all the information about submission process
assignment information	access to all the information about assignment process. A reviewer cannot be the author of the papers that are assigned to him
review information	access to all the information about review process
webAccess	access to the web site of the conference

Figure A.21: Dependency table $(C)D_t$

Task	Dependency
start up	start up information
submission	submission information user information paper information webAccess
paper partitioning	start up information partitioning information paper information user information
assignment papers	assignment information paper information user information
review process	review information paper information webAccess

Figure A.22: Task-Dependency table $(C)TD_t$

Function	Dependency
management user	user information submission information assignment information
management review	review information
management paper	paper information submission information assignment information review information partitioning information
management assignment	assignment information
management partitioning	partitioning information
management process	start up information
webSite	webAccess

Figure A.23: Function-Dependency table $(C)FD_t$

Figures A.24 and A.25 illustrate the dependencies and the their links with tasks at layer C+1.

Dependency	Description
new organisation	organisation is changed
election	start the election of vice-chairs
classification	it is necessary to start the classification of papers
partition	it is necessary to start the partitioning of papers

Figure A.24: Dependency table $(C + 1)D_t$

Task	Dependency
modify startup	+start up information new organisation classification + webAccess
create sub-committees	+ user information election + webAccess
Vice-Chairs elections	+user information election classification + webAccess
paper classification	classification + paper information partition + webAccess
partition paper	partition + paper information + partitioning information + webAccess

Figure A.25: Task-Dependency table $(C + 1)TD_t$

Topologies Tables, in turn, express the topological constraints over the environment. As a web-based system, our conference management system does not impose many topological constraints over the environment: in fact, we identify just one constraint — the locus where the functions are allocated. Figures A.26, A.27 and A.28 present the Topologies Tables, while Figure A.29 presents the Task-Topologies Table at layer C+1.

Topology	Description
place	this is the locus where the functions are allocated

Figure A.26: Topology table $(C)Top_t$

Task	Topology
start up	place
submission	place
paper partitioning	place
assignment papers	place
review process	place

Figure A.27: Task-Topology table $(C)TTop_t$

Function	Topology
management user	place
management review	place
management paper	place
management assignment	place
management partitioning	place
management process	place
webSite	place

Figure A.28: Function-Topology table $(C)FTop_t$

Task	Topology
modify start up	+place
create sub-commettes	+place
Vice-Chair elections	+place
paper classification	+place
partition paper	+place

Figure A.29: Task-Topology table $(C + 1)TTop_t$

A.4 From Analysis to Architectural Design

In order to link the Analysis step with the Architectural Design step, we relate the Analysis entities with the Architectural Design by means of Transition Tables. Figure A.30 shows

the mapping between tasks and roles: in this case, three different tasks are assigned to the same role (PC-chair). Since our system is composed of two layers, a similar set of tables will be defined also for the C+1 layer (Figure A.31).

Figure A.32 shows the mapping between resources and functions, Figures A.33 and A.34 show the mapping between dependencies and interactions respectively at layer C and at layer C+1, and finally Figure A.35 shows the mapping between topologies and workspaces.

Role	Task
PC-chair	paper partitioning start-up assignment papers
Author	submission
PC-member	review process

Figure A.30: Transition Role-Task table $(C)TRT_t$

Role	Task
ManagerStartUp	modify start up
Sub-Commette	create sub-commettees Vice-chair elections
Partitioner	papers classification partition papers

Figure A.31: Transition Role-Task table $(C + 1)TRT_t$

Resource	Function
People DB	management user
Paper DB	management paper management review management partitioning management assignment
Process DB	management process
WebService	webSite

Figure A.32: Transition Resource-Function table $(C)TRF_t$

Dependency	Interaction
start up information	
user information	User-Rule
paper information	Author-Rule
partitioning information	Match-Rule
submission information	Deadline-Rule
assignment information	AutRev-Rule Review-Rule
review information	Author-Rule
webAccess	Access-Rule

Figure A.33: Transition Interaction-Dependency table $(C)TID_t$

Dependency	Interaction
new organisation	Org-Rule
election	Vice-Rule
classification	Class-Rule
partition	Part-Rule

Figure A.34: Transition Interaction-Dependency table $(C + 1)TID_t$

Topology	Workspace
place	Wplace

Figure A.35: Transition Topology-Workspace table $(C)TTopW_t$

A.5 Architectural Design

The Architectural Design step consists of three sets of tables: Entities Tables, Interaction Tables and Topological Tables.

Entities Tables describe both the active entities (the roles) able to perform some action in the system, and the passive entities (the resources) which provide services.

The Zooming Table for the architectural design is reported in Figure A.36. In the table the entities derived from the in-zoom operation on role PC-Chair are listed. In particular the firsts three entities are sub-roles of PC-Chair, the underlined entities are interactions and the wave underlined entities are actions.

Layer C	Layer C+1
PC-chair	ManagerStartUp, Sub-Committee Partitioner, ... <u>Vice-Rule, Org-Rule</u> <u>Part-Rule, Class-Rule</u> <u>change information</u> <u>read paper information</u> <u>modify paper information</u> <u>define vice-chair</u>

Figure A.36: Architectural Zooming table $(C)Z_t$

Figures A.37, A.38, A.39 and A.40 report respectively the Action table and the Role-Action table for the layer C, and Action table and the Role-Action table for the layer C+1.

Action	Description
login	user authentication
send paper	user compiles form and sends his paper
publish deadline	user generates/modifies deadline
partition	user splits papers according to key words
assignment	user assigns papers
read paper	user reads papers
download paper	user download paper from the web
write review	user writes the review

Figure A.37: Action table $(C)A_t$

Role	Action
PC-chair	login, assignment publish deadline partition
Author	login, send paper
PC-member	login, read paper write review download paper

Figure A.38: Role-Action table $(C)RA_t$

Action	Description
change information	changing an information in start up process
read paper information	reading information about a paper
modify paper information	modifying an information about a paper
define vice-chair	election of Vice-Chair

Figure A.39: Role-Action table $(C + 1)A_t$

Role	Action
ManagerStartUp	+ login change information
Sub-Committee	+ login define vice-chair
Partitioner	+login + partitions read paper information modify paper information

Figure A.40: Role-Action table $(C + 1)RA_t$

Figures A.41 and A.42 report respectively the Operation table and the Resource-Operation table for the layer C.

Operation	Description
store paper	storing paper and its information
get paper	providing paper and its information
store user	storing user information
get user	providing user information
store process	storing process information
get process	providing process information
store assignment	storing assignment information
access web	friendly interface of the application
store review	storing review information

Figure A.41: Operation table $(C)O_t$

Resource	Operation
People DB	store user get user
Paper DB	store paper get paper store assignment store review
Process DB	get process store process
WebService	access Web

Figure A.42: Resource-Operation table $(C)RO_t$

Interaction Tables describe the interaction between roles and resources: more precisely, the Interaction table $((L)I_t)$ defines the single interactions, the Role-Interaction table $((L)RoI_t)$ specifies the interactions where each role is involved, and the Resource-Interaction table $((L)ReI_t)$ specifies the interactions where each resource is involved. Figure A.43 shows the organisational rules for conference management. For example, the AutRev-Rule specifies that an author cannot be the reviewer of his/her own paper, while the Author-Rule specifies that the author can access only the public information about his/her own paper even if he/she is a PC-member.

Interaction	Description
Deadline-Rule	send paper is possible if and only if time is minus then deadline submission
User-Rule	get user is possible if the request user is the requester or the requester is the PC-chair
Author-Rule	author can access and modify only his public paper information
Match-Rule	papers can be partitioned according key words
AutRev-Rule	the PC-member cannot be the author of paper
Review-Rule	the PC-member cannot access to private information about his papers
Access-Rule	the access to the system must be authorised

Figure A.43: Interaction table $(C)I_t$

Figures A.44 and A.45 depict the links among roles and interactions and among re-

sources and interactions.

Role	Interaction
PC-chair	Deadline-Rule User-Rule Author-Rule Match-Rule AutRev-Rule Review-Rule Access-Rule
Author	Deadline-Rule User-Rule Author-Rule Access-Rule
PC-member	User-Rule Author-Rule AutRev-Rule Review-Rule Access-Rule

Figure A.44: Role-Interaction table $(C)RoI_t$

Resource	Interaction
People DB	User-Rule Match-Rule
Paper DB	Author-Rule AutRev-Rule Review-Rule
Process DB	Deadline-Rule
WebService	

Figure A.45: Resource-Interaction table $(C)ReI_t$

Figures A.46 and A.47 depict respectively the Interaction table and the Role- Interaction Table at layer C+1.

Interaction	Description
Org-Rule	if the organisation is changed then start Sub-Commette
Vice-Rule	the Vice-Chair must be an expert of the filed
Class-Rule	a paper can belong only at one class
Part-Rule	papers can be partitioned according their classification

Figure A.46: Interaction table $(C + 1)I_t$

Role	Interaction
ManagerStartUp	+Deadline-Rule +User-Rule +Access-Rule Org-Rule
Sub-Committee	+User-Rule +Access-Rule Vice-Rule
Partitioner	+User-Rule +Match-Rule +Access-Rule Class-Rule Part-Rule

Figure A.47: Role-Interaction table $(C + 1)RoI_t$

Finally, Topological Tables describe the logical structure of the environment. As outlined above, the conference management system does not present a complex topology: so, a single workspace is enough for representing the logical structure of the environment (Figures A.48, A.50, A.49, A.51 and A.52).

Workspace	Description
Wplace	this is the workspace where the resources are be allocated

Figure A.48: Workspace table $(C)W_t$

Workspace	Resource
Wplace	People DB Paper DB Process DB WebService

Figure A.49: Workspace-Resource table $(C)WR_{e_t}$

Workspace	Connection
Wplace	

Figure A.50: Workspace-Connection table $(C)WC_t$

Role	Workspace
PC-Chair	Wplace
Author	Wplace
PC-member	Wplace

Figure A.51: Workspace-Role table $(C)WR_{o_t}$

Role	Workspace
ManagerStartUp	+Wplace
Sub-Commette	+Wplace
Partitioner	+Wplace

Figure A.52: Workspace-Role table $(C + 1)WR_{o_t}$

A.6 From Architectural Design to Detailed Design

Now, in order to transit to the Detailed Design phase, the “carving operation” has to be performed. Figure A.53 - a) shows the key layers of our system. Since the three roles at layer C derive from the same role (“Organisation”) at layer C-1, the carving operation is very simple: the three roles at layer C become agents, and the role at layer C-1 becomes a society (Figure A.53 - b). The result is the set of Mapping Tables, which relate the Architectural Design entities to the Detailed Design.

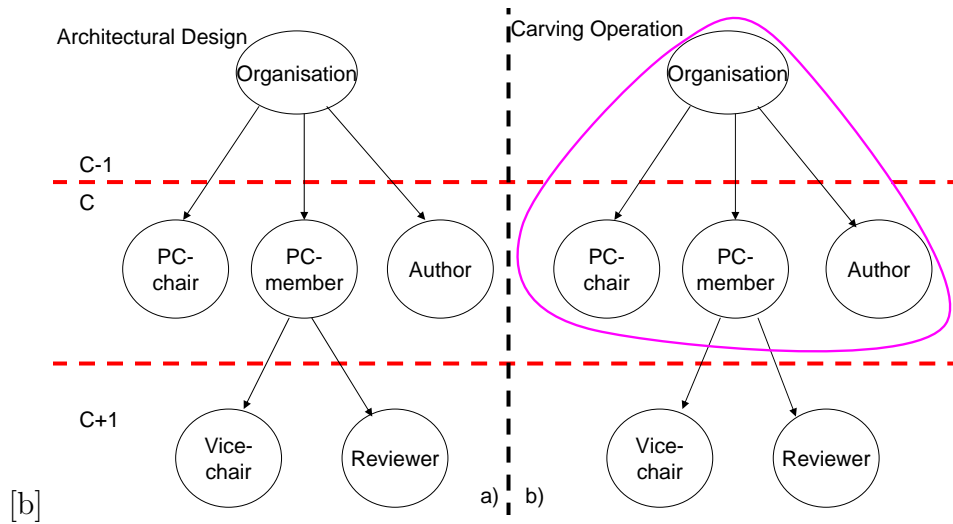


Figure A.53: Carving operation in the Conference Management

In particular, the Mapping Agent-Role table $((L)MAR_t)$ maps roles onto agents, the Mapping Artifact-Resource table $((L)MArR_t)$ maps resources onto artifacts, and the Mapping Artifact-Interaction table $((L)MArI_t)$ maps the rules specified in the Architectural Design onto the artifacts that implement and enforce them. In conference management, each role is assigned to a specific agent (Figure A.54) and each resource is assigned to a specific artifact (Figure A.55); on the contrary, some interactions could be mapped onto the same artifact (Figure A.56), as in the case of the Author-Rule and the Match-Rule, because both rules are used in the partitioning process.

Agent	Role
PC-Chair Agent	PC-chair
Author Agent	Author
PC-Member Agent	PC-member

Figure A.54: Mapping Agent-Role table $(C)MAR_t$

Artifact	Resource
Paper Artifact	Paper DB
People Artifact	People DB
Process Artifact	Process DB
Web Artifact	WebService

Figure A.55: Mapping Artifact-Resource table $(C)MArR_t$

Interaction	Artifact
Deadline-Rule	StartUp Artifact
User-Rule	User-Rule Artifact
Access-Rule	User-Rule Artifact
Author-Rule	Partition Artifact
Match-Rule	Partition Artifact
AutRev-Rule	Assignment Artifact
Review-Rule	Review Artifact

Figure A.56: Mapping Artifact-Interaction table $(C)MArI_t$

A.7 Detailed Design

The Detailed Design step exploits two sets of tables: Agent/Society Design Tables, and Environment Design Tables. The first set of tables depicts agents, individual artifacts, and the agent societies derived from the carving operation. Figures A.57, A.58 and A.59 report the Agent/Society Design Tables.

Agent	Artifact
PC-Chair Agent	PC-Chair Artifact
Author Agent	Author Artifact
PC-Member Agent	PC-Member Artifact

Figure A.57: Agent-Artifact table $(C)AA_t$

Society	Agent
Org	PC-Chair Agent Author Agent PC-Member Agent

Figure A.58: Society-Agent table $(C)SA_t$

Society	Artifact
Org	StartUp Artifact User-Rule Artifact Partitioning Artifact Assignment Artifact Review Artifact

Figure A.59: Society-Artifact table $(C)SAr_t$

In turn, Environment Design Tables concern the design of artifacts and workspaces: the Artifact-UsageInterface table (Figure A.60) details the operations provided by each artifact, the Aggregate-Artifact table (Figure A.61) – that is empty in this example – specifies which artifacts are part of the Aggregate generated by the carving, while the Workspace-Artifact table (Figure A.62) specifies the location of artifacts in the workspace.

Artifact	Usage Interface
PC-Chair Artifact	read start up information modify start up information login partition assignment
Author Artifact	login send paper
PC-Member Artifact	login read paper write review download paper
People Artifact	store user get user
Paper Artifact	store paper get paper store assignment store review
Process Artifact	get process store process
Web Artifact	access Web
StartUp Artifact	deadline extension update rule read rule
User-Rule Artifact	get user modify user
Partition Artifact	partition paper access classification
Assignment Artifact	check authors check reviewer
Review Artifact	check access to review information

Figure A.60: Artifact-UsageInterface table $(C)AUI_t$

Aggregate	Artifact

Figure A.61: Aggregate-Artifact table $(C)AggA_t$

Workspace	Artifact
Wplace	PC-Chair Artifact, Author Artifact PC-Member Artifact, People Artifact Process Artifact, Web Artifact StartUp Artifact, User-Rule Artifact Partition Artifact, Assignment Artifact Review Artifact, Paper Artifact

Figure A.62: Workspace-Artifact table $(C)WA_t$

The system design at layer C is now complete. Of course, this is still quite a high-level system view: several in-zoom operations are needed, until the system design is refined enough for implementation purposes.

Bibliography

- [1] ADELFE. http://www.pa.icar.cnr.it/~cossentino/FIPAmeth/docs/adelfe_july05.pdf.
- [2] ADELFE Group. ADELFE home page. <http://www.irit.fr/ADELFE/>.
- [3] aliCE Research Group. SODA home page. <http://soda.alice.unibo.it>.
- [4] Stefania Bandini, Sara Manzoni, and Carla Simone. Supporting the application of situated cellular agents in non-uniform spaces. *Future Generation Computer Systems*, 21(4):627–631, 2005.
- [5] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice, Second Edition*. Addison-Wesley, 2003.
- [6] Fabio Bellifemine, Agostino Poggi, and Giovanni Rimassa. Developing multi-agent systems with a fipa-compliant agent framework. *Software-Practice & Experience*, 31(2):103–128, 2001.
- [7] Fabio Bellifemine, Agostino Poggi, and Giovanni Rimassa. Jade: a fipa2000 compliant agent development environment. In *AGENTS '01: Proceedings of the fifth international conference on Autonomous agents*, pages 216–217, New York, NY, USA, 2001. ACM.
- [8] Federico Bergenti, Marie-Pierre Gleizes, and Franco Zambonelli, editors. *Methodologies and Software Engineering for Agent Systems: The Agent-Oriented Software Engineering Handbook*, volume 11 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*. Kluwer Academic Publishers, jun 2004.
- [9] Carole Bernon, Valérie Camps, Marie-Pierre Gleizes, and Gauthier Picard. Engineering adaptive multi-agent systems: The ADELFE methodology. In Henderson-Sellers and Giorgini [87], chapter VII, pages 172–202.
- [10] Carole Bernon, Massimo Cossentino, Marie Pierre Gleizes, Paola Turci, and Franco Zambonelli. A study of some multi-agent meta-models. In James Odell, Paolo Giorgini, and Jörg P. Müller, editors, *Agent Oriented Software Engineering V*, volume 3382 of *Lecture Notes in Computer Science*, pages 62–77. Springer, 2004.
- [11] Nino Boccara. *Modeling Complex Systems (Graduate Texts in Contemporary Physics)*. Springer, 2003.
- [12] G. Booch. *Object-oriented analysis and design with applications*. Addison Wesley, 1994.
- [13] Paolo Bresciani, Paolo Giorgini, Fausto Giunchiglia, John Mylopoulos, and Anna Perini. Tropos: An agent-oriented software development methodology. *Autonomous Agent and Multi-Agent Systems*, 8(3):203–236, May 2004.
- [14] S. Brinkkemper, K. Lyytinen, and R. Welke. *Method engineering: Principles of method construction and tool support*. Kluwer Academic Publishers, 1996.

- [15] Sjaak Brinkkemper. Method engineering: engineering of information systems development methods and tools. *Information & Software Technology*, 38(4):275–280, 1996.
- [16] Sjaak Brinkkemper, Motoshi Saeki, and Frank Harmsen. Meta-modelling based assembly techniques for situational method engineering. *Information Systems*, 24(3):209–228, 1999.
- [17] Stefan Bussmann. Agent-oriented programming of manufacturing control tasks. In *ICMAS '98: Proceedings of the 3rd International Conference on Multi Agent Systems*, pages 57–63, Washington, DC, USA, 1998. IEEE Computer Society.
- [18] Roberto Caico, Massimo Cossentino, Luca Sabatucci, Valeria Seidita, and Salvatore Gaglio. Metameth: a tool for process definition and execution. In Flavio De Paoli, Antonella Di Stefano, Andrea Omicini, and Corrado Santoro, editors, *WOA*, volume 204 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2006.
- [19] Giovanni Caire, Wim Coulier, Francisco Garijo, Jorge Gòmez-Sanz, Juan Pavòn, Paul Kearney, and Philippe Massonet. The MESSAGE methodology. In Bergenti et al. [8], chapter 9, pages 177–194.
- [20] Giovanni Caire, Wim Coulier, Francisco J. Garijo, Jorge Gomez, Juan Pavòn, Francisco Leal, Paulo Chainho, Paul E. Kearney, Jamie Stark, Richard Evans, and Philippe Massonet. Agent oriented analysis using Message/UML. In Michael Wooldridge, Gerhard Weiss, and Paolo Ciancarini, editors, *Agent-Oriented Software Engineering II*, volume 2222 of *LNCS*, pages 119–135. Springer, 2002. 2nd International Workshop (AOSE 2001), Montreal, Canada, 29 May 2001. Revised Papers and Invited Contributions.
- [21] Cosmin Carabelea, Olivier Boissier, and Cristiano Castelfranchi. Using social power to enable agents to reason about being part of a group. In Marie Pierre Gleizes, Andrea Omicini, and Franco Zambonelli, editors, *Engineering Societies in the Agents World V*, volume 3451 of *Lecture Notes in Computer Science*, pages 166–177. Springer, 2005.
- [22] Luca Cardelli. Abstractions for mobile computation. In Jan Vitek and Christian D. Jensen, editors, *Secure Internet Programming*, volume 1603 of *Lecture Notes in Computer Science*, pages 51–94. Springer, 1999.
- [23] **CARTAgO**. Home page. <http://www.alice.unibo.it:16080/projects/cartago/>.
- [24] Cristiano Castelfranchi. Modelling social action for ai agents. *Artificial Intelligence*, 103(1-2):157–182, 1998.
- [25] Cristiano Castelfranchi and W. Lewis Johnson, editors. *Proceedings of the 1st International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2002)*, Bologna, Italy, 15–19 July 2002. ACM Press.
- [26] Luca Cernuzzi, Massimo Cossentino, and Franco Zambonelli. Process models for agent-based development. *Engineering Applications of Artificial Intelligence*, 18(2):205–222, March 2005.

- [27] Luca Cernuzzi and Gustavo Rossi. On the evaluation of agent oriented methodologies. In J. Debenham, B. Henderson-Sellers, N. Jennings, and J. Odell, editors, *Proceedings of the OOPSLA 2002 Workshop on Agent-Oriented Methodologies*, 2002.
- [28] Luca Cernuzzi and Franco Zambonelli. Dealing with adaptive multiagent systems organisations in the Gaia methodology. In Jörg P. Müller and Franco Zambonelli, editors, *Agent-Oriented Software Engineering VI*, volume 3950 of *LNCS*, pages 109–123. Springer, 2006. 6th International Workshop (AOSE 2005), Utrecht, The Netherlands, 25–26 July 2005. Revised and Invited Papers.
- [29] John Charles. Middleware moves to the forefront. *Computer*, 32(5):17–19, 1999.
- [30] Paolo Ciancarini, Oscar Nierstrasz, and Robert Tolksdorf. A case study in coordination: Conference management on Internet. <ftp://ftp.cs.unibo.it/pub/cianca/coordina.ps.gz>, 1996.
- [31] Paolo Ciancarini and Alexander L. Wolf, editors. *Coordination Languages and Models, 3rd International Conference*, volume 1594 of *LNCS*. Springer, 1999.
- [32] Derek Coleman, Patrick Arnold, Stephanie Bodoff, Chris Dollin, Helena Gilchrist, Fiona Hayes, and Paul Jeremaes. *Object-Oriented Development. The Fusion Method*. Prentice-Hall, 1994.
- [33] Anne Collinot and Alexis Drogoul. Using the cassiopeia method to design a robot soccer team. *Applied Artificial Intelligence*, 12(2–3):127–147, 1998.
- [34] Confious. Home page. <http://confioussite.ics.forth.gr/>.
- [35] Massimo Cossentino. From requirements to code with the PASSI methodology. In Henderson-Sellers and Giorgini [87], chapter IV, pages 79–106.
- [36] Massimo Cossentino, Salvatore Gaglio, Alfredo Garro, and Valeria Seidita. Method fragments for agent design methodologies: from standardisation to research. *International Journal of Agent Oriented Software Engineering*, 1(1):91–121, 2007.
- [37] Massimo Cossentino, Nicolas Gaud, Stéphane Galland, Vincent Hilaire, and Abder Koukam. A holonic metamodel for agent-oriented analysis and design. In Vladimír Mavřík, Valeriy Vyatkin, and Armando W. Colombo, editors, *Holonic and Multi-Agent Systems for Manufacturing*, volume 4659 of *Lecture Notes in Computer Science*, pages 237–246. Springer, 2007. Holonic and Multi-Agent Systems for Manufacturing, Third International Conference on Industrial Applications of Holonic and Multi-Agent Systems, HoloMAS 2007, Regensburg, Germany, September 3-5, 2007, Proceedings.
- [38] Massimo Cossentino, Luca Sabatucci, and Antonio Chella. Patterns reuse in the PASSI methodology. In Andrea Omicini, Paolo Petta, and Jeremy Pitt, editors, *Engineering Societies in the Agents World IV*, volume 3071 of *LNAI*, pages 294–310. Springer-Verlag, June 2004. 4th International Workshop (ESAW 2003), London, UK, 29–31 October 2003. Revised Selected and Invited Papers.

- [39] Khanh Hoa Dam and Michael Winikoff. Comparing agent-oriented methodologies. In Paolo Giorgini, Brian Henderson-Sellers, and Michael Winikoff, editors, *Agent-Oriented Information Systems*, volume 3030 of *LNCS*, pages 78–93. Springer, 2004. 5th International Bi-Conference Workshop, Agent-Oriented Information Systems 2003, Melbourne, Australia, July 14, 2003 and Chicago, IL, USA, October 13th, 2003, Revised Selected Papers.
- [40] Mehdi Dastani, Joris Hulstijn, Frank Dignum, and John-Jules Ch. Meyer. Issues in multiagent system development. In Castelfranchi and Johnson [25], pages 922–929.
- [41] Scott A. DeLoach. Engineering organization-based multiagent systems. In Garcia [68], pages 109–125.
- [42] Scott A. DeLoach. Engineering organization-based multiagent systems. In Alessandro F. Garcia, Ricardo Choren, Carlos José Pereira de Lucena, Paolo Giorgini, Tom Holvoet, and Alexander B. Romanovsky, editors, *Software Engineering for Multi-Agent Systems IV, Research Issues and Practical Applications*, volume 3914 of *LNCS*, pages 109–125. Springer, 2006. 4th International Workshop on Software Engineering for Large-Scale Multi-Agent Systems (SELMAS 2005), St. Louis, Missouri, USA, 15-16 May 2005. Revised Selected Papers.
- [43] Scott A. DeLoach. Developing a multiagent conference management system using the o-mase process framework. In *8th International Workshop on Agent Oriented Software Engineering*, Honolulu, Hawaii, USA, 14 May 2007. Invited paper.
- [44] Scott A. DeLoach and Madhukar Kumar. Multi-agent systems engineering: An overview and case study. In Henderson-Sellers and Giorgini [87], chapter XI, pages 317–340.
- [45] Scott A. DeLoach and Jorge L. Valenzuela. An agent-environment interaction model. In Lin Padgham and Franco Zambonelli, editors, *Agent-Oriented Software Engineering VII*, volume 4405 of *LNCS*. Springer, 2007. 7th International Workshop (AOSE 2006), Hakodate, Japan, 8 May 2006. Selected Revised and Invited Papers.
- [46] Enrico Denti, Antonio Natali, and Andrea Omicini. On the expressive power of a language for programming coordination media. In *1998 ACM Symposium on Applied Computing (SAC'98)*, pages 169–177, Atlanta, GA, USA, 27 February – 1 March 1998. ACM. Special Track on Coordination Models, Languages and Applications.
- [47] Enrico Denti and Andrea Omicini. Designing multi-agent systems around a programmable communication abstraction. In John-Jules Ch. Meyer and Pierre-Yves Schobbens, editors, *Formal Models of Agents*, volume 1760 of *LNAI*, pages 90–102. Springer-Verlag, 1999. ESPRIT Project ModelAge Final Workshop, Selected Papers.
- [48] Enrico Denti, Andrea Omicini, and Alessandro Ricci. Coordination tools for MAS development and deployment. *Applied Artificial Intelligence*, 16(9–10):721–752, October–December 2002. Special Issue: Engineering Agent Systems – Best of “From Agent Theory to Agent Implementation (AT2AI-3)”.

- [49] Developers. Prometheus home page. <http://www.cs.rmit.edu.au/agents/SAC2/methodology.html>.
- [50] Giovanna Di Marzo Surugendo, Marie-Pierre Gleizes, and Anthony Karageorgos. Self-organisation and emergence in mas: An overview. *Informatica*, 30(1):45–54, 2006.
- [51] Frank Dignum, Virginia Dignum, Sven Koenig, Sarit Kraus, Munindar P. Singh, and Michael Wooldridge, editors. *Proceedings of Autonomous Agents and Multiagent Systems 2005*. ACM Press, 25–29 July 2005.
- [52] Virginia Dignum. *A Model for Organizational Interaction, based on Agents, founded in Logic*. PhD thesis, University of Utrecht, 2003.
- [53] Dov Dori. *Object-Process Methodology: A Holistic System Paradigm*. Springer, 2002.
- [54] Dov Dori and Iris Reinhartz-Berger. An opm-based metamodel of system development process. In Il-Yeol Song, Stephen W. Liddle, Tok Wang Ling, and Peter Scheuermann, editors, *ER*, volume 2813 of *Lecture Notes in Computer Science*, pages 105–117. Springer, 2003.
- [55] Timon C. Du, Eldon Y. Li, and An-Pin Chang. Mobile agents in distributed network management. *Communication ACM*, 46(7):127–132, 2003.
- [56] Francisco Durán and Alberto Verdejo. A conference reviewing system in Mobile Maude. In Fabio Gadducci and et al., editors, *4th International Workshop on Rewriting Logic and its Applications*, volume 71 of *ENTCS*, pages 79–95. Elsevier, 2002.
- [57] EDAS. Home page. <http://edas.info/>.
- [58] Mark Edwards. A brief history of holons. <http://www.integralworld.net/index.html?edwards13.html>.
- [59] Niles Eldredge. *Unfinished Synthesis: Biological Hierarchies and Modern Evolutionary Thought*. Oxford University Press, 1985.
- [60] EROOS. Home page. <http://www.cs.kuleuven.ac.be/cwis/research/som/EROOS/>.
- [61] Marc Esteva, Bruno Rosell, Juan A. Rodríguez-Aguilar, and Josep Lluís Arcos. AMELI: An agent-based middleware for electronic institutions. In Nicholas R. Jennings, Carles Sierra, Liz Sonenberg, and Milind Tambe, editors, *3rd international Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2004)*, volume 1, pages 236–243, New York, USA, 19–23 July 2004. IEEE Computer Society.
- [62] FIPA. Fipa-acl. <http://www.fipa.org/specs/fipa00061/index.html>.
- [63] FIPA. Methodology home page. <http://www.pa.icar.cnr.it/~cossentino/FIPAmeth/>.
- [64] FIPA Group. AUML home page. <http://www.auml.org>.
- [65] Ian Foster and Carl Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, 1999.

- [66] Foundation for Intelligent Physical Agents. FIPA home page. <http://www.fipa.org>.
- [67] Alfonso Fuggetta. Software process: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 25–34, New York, NY, USA, 2000. ACM Press.
- [68] Alessandro F. et al. Garcia, editor. *Software Engineering for Multi-Agent Systems IV, Research Issues and Practical Applications*, volume 3914 of *LNCS*. Springer, 2006.
- [69] Francisco J. Garijo, Jorge J. Gómez-Sanz, and Philippe Massonet. The MESSAGE methodology for agent-oriented analysis and design. In Henderson-Sellers and Giorgini [87], chapter VIII, pages 203–235.
- [70] Les Gasser. MAS infrastructure: Definitions, needs and prospects. In Thomas Wagner and Omer F. Rana, editors, *Agents Workshop on Infrastructure for Multi-Agent Systems*, volume 1887 of *Lecture Notes in Computer Science*, pages 1–11. Springer, 2001. International Workshop on Infrastructure for Multi-Agent Systems, Barcelona, Spain, June 3-7, 2000, Revised Papers.
- [71] David Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.
- [72] David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2):97–107, February 1992.
- [73] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamental of Software Engineering*. Prentice Hall, second edition, 2002.
- [74] Joseph Andrew Giampapa and Katia Sycara. Team-oriented agent coordination in the RETSINA multi-agent system. Technical Report CMU-RI-TR-02-34, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, December 200. Presented at AAMAS 2002 Workshop on Teamwork and Coalition Formation.
- [75] Kathleen R. Gibson and Tim Ingold, editors. *Tools, Language & Cognition in Human Evolution*. Cambridge University Press, 1993.
- [76] Paolo Giorgini, Manuel Kolp, and John Mylopoulos. Multi-agent and software architectures: A comparative case study. In Fausto Giunchiglia, James Odell, and Gerhard Weiß, editors, *Agent-Oriented Software Engineering III*, volume 2585 of *LNCS*, pages 101–112. Springer, 2003. Third International Workshop, AOSE 2002, Bologna, Italy, July 15, 2002, Revised Papers and Invited Contributions.
- [77] Paolo Giorgini, Manuel Kolp, John Mylopoulos, and Jaelson Castro. Tropos: A requirements-driven methodology for agent-oriented software. In Henderson-Sellers and Giorgini [87], chapter II, pages 20–45.
- [78] Adriana Giret and Vicente Botti. Holons and agents. *Journal of Intelligent Manufacturing*, 15(5):645–659, Nov 2004.

- [79] Adriana Giret, Vicente J. Botti, and Soledad Valero. Mas methodology for hms. In Vladimír Marík, Robert W. Brennan, and Michal Pechoucek, editors, *Holonic and Multi-Agent Systems for Manufacturing*, volume 3593 of *Lecture Notes in Computer Science*, pages 39–49. Springer, 2005. Holonic and Multi-Agent Systems for Manufacturing, Second International Conference on Industrial Applications, of Holonic and Multi-Agent Systems, HoloMAS 2005, Copenhagen, Denmark, August 22–24, 2005, Proceedings.
- [80] Cesar Gonzalez-Perez, Tom McBride, and Brian Henderson-Sellers. A metamodel for assessable software development methodologies. *Software Quality Journal*, 13(2):195–214, 2005.
- [81] Grasia Group. INGENIAS home page. <http://grasia.fdi.ucm.es/ingenias/index.php>.
- [82] Marjorie J. Grene. Hierarchies in biology. *American Scientist*, 75:504–510, 1987.
- [83] John A. Hamilton, Jr. and Udo W. Pooch. A survey of object-oriented methodologies. In *Conference on TRI-Ada '95: Ada's role in global markets: solutions for a changing complex world*, pages 226–234, New York, NY, USA, 1995. ACM Press.
- [84] Brian Henderson-Sellers. Method engineering for object oriented systems development. *Communication ACM*, 46(10):73–78, 2003.
- [85] Brian Henderson-Sellers. Creating a comprehensive agent-oriented methodology: Using method engineering and the OPEN metamodel. In Henderson-Sellers and Giorgini [87], chapter XIII, pages 368–397.
- [86] Brian Henderson-Sellers. Evaluating the feasibility of method engineering for the creation of agent-oriented methodologies. In Michal Pechoucek, Paolo Petta, and László Zsolt Varga, editors, *Multi-Agent Systems and Applications IV, 4th International Central and Eastern European Conference on Multi-Agent Systems, CEEMAS 2005, Budapest, Hungary, September 15-17, 2005, Proceedings*, volume 3690 of *Lecture Notes in Computer Science*, pages 142–152. Springer, 2005.
- [87] Brian Henderson-Sellers and Paolo Giorgini, editors. *Agent Oriented Methodologies*. Idea Group Publishing, Hershey, PA, USA, June 2005.
- [88] Brian Henderson-Sellers and Cesar Gonzalez-Perez. A comparison of four process metamodels and the creation of a new generic standard. *Information & Software Technology*, 47(1):49–65, 2005.
- [89] Gordon W. Hewes. A history of speculation on the relation between tools and languages. In Gibson and Ingold [75], pages 20–31.
- [90] Francis Heylighen, Paul Cilliers, and Carlos Gershenson. Complexity and philosophy. *Computing Research Repository*, abs/cs/0604072, 2006.
- [91] HMS. Holonic manufacturing systems. <http://hms.ifw.uni-hannover.de/index.htm>.

- [92] Marc-Philippe Huget. Nemo: an agent-oriented software engineering methodology. In J. Debenham, B. Henderson-Sellers, N. Jennings, and J. Odell, editors, *Proceedings of the OOPSLA 2002 Workshop on Agent-Oriented Methodologies*, 2002.
- [93] M.E.C. Hull, P.S. Taylor, J.R.P. Hanna, and R.J. Millar. Software development processes - an assessment. *Information and Software Technology*, 44:1–12, January 2002.
- [94] IBM. WebSphere home page. www-3.ibm.com/software/ts/mqseries.
- [95] Carlos A. Iglesias and Mercedes Garijo. Agent-oriented methodology MAS-CommonKADS. In Henderson-Sellers and Giorgini [87], chapter III, pages 46–78.
- [96] INGENIAS. Home page. <http://grasia.fdi.ucm.es/ingenias/>.
- [97] JADE. Home page. <http://jade.tilab.com/>.
- [98] Nicholas R. Jennings. Agent-oriented software engineering. In Ibrahim F. Imam, Yves Kodratoff, Ayman El-Dessouki, and Moonis Ali, editors, *IEA/AIE*, volume 1611 of *LNCS*, pages 4–10. Springer, 1999. 12th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems, IEA/AIE-99, Cairo, Egypt, May 31 – June 3, 1999, Proceedings.
- [99] Nicholas R. Jennings. An agent-based approach for building complex software systems. *Communication ACM*, 44(4):35–41, 2001.
- [100] Frederick P. Brooks Jr. No silver bullet – essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19, 1987.
- [101] Frederick P. Brooks Jr. *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition*. Addison-Wesley Professional, 2nd edition edition, Aug 1995.
- [102] Stanley M. Sutton Jr. Middleware selection. In Wolfgang Emmerich and Stefan Tai, editors, *Engineering Distributed Objects, EDO 2000*, volume 1999 of *Lecture Notes in Computer Science*, pages 2–7. Springer, 2001. Second International Workshop, EDO 2000, Davis, CA, USA, November 2-3, 2000, Revised Papers.
- [103] Thomas Juan, Adrian R. Pearce, and Leon Sterling. ROADMAP: extending the Gaia methodology for complex open systems. In Castelfranchi and Johnson [25], pages 3–10.
- [104] Elizabeth A. Kendall, Margaret T. Malkoun, and Chong H. Jiang. A methodology for developing agent based systems. In Chengqi Zhang and Dickson Lukose, editors, *DAI*, volume 1087 of *LNCS*, pages 85–99. Springer, 1996. Distributed Artificial Intelligence: Architecture and Modelling, First Australian Workshop on DAI, Canberra, ACT, Australia, November 13, 1995, Proceedings.
- [105] Jeffrey Kephart. Software agents and the route to the information economy. *Proceedings of National Academy of Science*, 99:7207–7213, May 2002.

- [106] David Kinny, Michael P. Georgeff, and Anand S. Rao. A methodology and modelling technique for systems of bdi agents. In Walter Van de Velde and John W. Perram, editors, *MAAMAW*, volume 1038 of *LNCIS*, pages 56–71. Springer, 1996. 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World, Eindhoven, The Netherlands, January 22-25, 1996, Proceedings.
- [107] Arthur Koestler. *The ghost in the machine*. Arkana, 1989.
- [108] Manuel Kolp, Paolo Giorgini, and John Mylopoulos. Organizational multi-agent architectures: a mobile robot example. In *AAMAS '02: Proceedings of the first international joint conference on Autonomous agents and multiagent systems*, pages 94–95, New York, NY, USA, 2002. ACM.
- [109] Jacek Kopecký, Dumitru Roman, Matthew Moran, and Dieter Fensel. Semantic web services grounding. *aict-iciw*, 0:127, 19–25 February 2006. International Conference on Internet and Web Applications and International Conference on Internet and Web Applications and Services.
- [110] Philippe Kruchten. *The Rational Unified Process: An Introduction*. Addison-Wesley Professional, 3rd edition, December 2003.
- [111] Timothy P. Layton. *Information Security: Design, Implementation, Measurement, and Compliance*. AUERBACH, Grover, Missouri, USA, 1st edition, July 2006.
- [112] Jurgen Lind. *Iterative Software Engineering for Multiagent Systems: The Massive Method*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2001.
- [113] Anna Liu and Ian Gorton. Accelerating cost middleware acquisition: The i-mate process. *IEEE Software*, 20(2):72–79, 2003.
- [114] Marco Mamei and Franco Zambonelli. Programming stigmergic coordination with the TOTA middleware. In Dignum et al. [51], pages 415–422.
- [115] Marco Mamei and Franco Zambonelli. Programming modular robots with the tota middleware. In Hideyuki Nakashima, Michael P. Wellman, Gerhard Weiss, and Peter Stone, editors, *5th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2006)*, pages 485–487. ACM, 2006.
- [116] Vladimír Marík and Michal Pechoucek. Holons & agents: Recent development and mutual impacts. In *DEXA '01: Proceedings of the 12th International Workshop on Database and Expert Systems Applications*, pages 605–607, Washington, DC, USA, 2001. IEEE Computer Society.
- [117] G. Jason Mathews and Barry E. Jacobs. Electronic management of the peer review process. In *5th International WWW conference on Computer networks and ISDN systems*, pages 1523–1538. Elsevier Science Publishers, 1996.

- [118] Emanuela Mattiolo. Ingegnerizzazione di applicazioni con metodologie agent oriented: un caso applicativo. <http://www.alice.unibo.it/alice/Controller?area=Theses&id=28>, 2006.
- [119] MEnSA group. MEnSA home page. <http://www.mensa-project.org/download.php>.
- [120] Mark Merkow and James Breithaupt. *Information Security: Principles and Practices*. Security Series. Prentice Hall, August 2005.
- [121] Methodology Working Group. IEEE-FIPA methodology working group home page. <http://www.fipa.org/activities/methodology.html>.
- [122] Microsoft. .Net home page. www.microsoft.com/net.
- [123] Ambra Molesini, Enrico Denti, and Andrea Omicini. MAS meta-models on test: UML vs. OPM in the SODA case study. In Michal Pěchouček, Paolo Petta, and László Zolt Varga, editors, *Multi-Agent Systems and Applications IV*, volume 3690 of *LNAI*, pages 163–172. Springer, 2005. 4th International Central and Eastern European Conference on Multi-Agent Systems (CEEMAS’05), Budapest, Hungary, 15–17 September 2005, Proceedings.
- [124] Ambra Molesini, Enrico Denti, and Andrea Omicini. From AOSE methodologies to MAS infrastructures: The SODA case study. In *8th International Workshop “Engineering Societies in the Agents World” (ESAW 2007)*, 2007.
- [125] Ambra Molesini, Alessandro Garcia, Christina Chavez, and Thais Batista. On the interplay of crosscutting and mas-specific styles. In Flavio Oquendo, editor, *Software Architecture*, volume 4758 of *Lecture Notes in Computer Science*, pages 317–320. Springer Berlin, September 2007. First European Conference, ECSA 2007 Aranjuez, Spain, September 24–26, 2007. Proceedings.
- [126] Ambra Molesini, Alessandro Garcia, Christina Chavez, and Thais Batista. On the quantitative analysis of architecture stability in aspectual composition. 2008. Accepted to Working IEEE/IFIP Conference on Software Architecture, WICSA 2008 18-21 February 2008, Vancouver, BC, Canada.
- [127] Ambra Molesini, Andrea Omicini, Enrico Denti, and Alessandro Ricci. SODA: A roadmap to artefacts. In Oğuz Dikenelli, Marie-Pierre Gleizes, and Alessandro Ricci, editors, *Engineering Societies in the Agents World VI*, volume 3963 of *LNAI*, pages 49–62. Springer, June 2006. 6th International Workshop (ESAW 2005), Kuşadası, Aydın, Turkey, 26–28 October 2005. Revised, Selected & Invited Papers.
- [128] Ambra Molesini, Andrea Omicini, Alessandro Ricci, and Enrico Denti. Zooming multi-agent systems. In Jörg P. Müller and Franco Zambonelli, editors, *Agent-Oriented Software Engineering VI*, volume 3950 of *LNCS*, pages 81–93. Springer, 2006. 6th International Workshop (AOSE 2005), Utrecht, The Netherlands, 25–26 July 2005. Revised and Invited Papers.

- [129] Ambra Molesini, Andrea Omicini, and Mirko Viroli. Environment in Agent-Oriented Software Engineering methodologies. *Multiagent and Grid Systems*, 4, 2008. Special Issue on Environment Engineering for MAS.
- [130] Sara Montagna, Alessandro Ricci, and Andrea Omicini. A&A for modelling and engineering simulations in Systems Biology. *International Journal of Agent-Oriented Software Engineering*, 2(2), 2008. Special Issue on Multi-Agent Systems and Simulation.
- [131] Carlo Montangero and Laura Semini. Composing specifications for coordination. In Ciancarini and Wolf [31], pages 118–133.
- [132] Harold J. Morowitz. *The Mind, the Brain, and Complex Adaptive Systems: Proceedings (Santa Fe Institute Studies in the Sciences of Complexity Proceedings)*. Addison Wesley Publishing Company, jan 1995.
- [133] Haralambos Mouratidis, Paolo Giorgini, and Gordon A. Manson. Modelling secure multiagent systems. In *Proceedings of the Second International Joint Conference on Autonomous Agents & Multiagent Systems*, pages 859–866. ACM, 2003. The Second International Joint Conference on Autonomous Agents & Multiagent Systems, AAMAS 2003, July 14–18, 2003, Melbourne, Victoria, Australia.
- [134] F.I. Moxley. On the specification of complex software systems. In *Engineering of Complex Computer Systems, 1996. Proceedings.*, pages 134–138, 21–25 October 1996. Second IEEE International Conference on Engineering of Complex Computer Systems. Montreal, Que., Canada.
- [135] Multiagent & Cooperative Robotics Laboratory. MaSE home page. <http://macr.cis.ksu.edu/projects/mase.htm>.
- [136] MyReview. Home page. <http://myreview.lri.fr/>.
- [137] B.A. Nardi. *Context and Consciousness: Activity Theory and Human-Computer Interaction*. MIT Press, 1996.
- [138] Elena Nardini, Ambra Molesini, Andrea Omicini, and Enrico Denti. SPEM on test: the SODA case study. In *23th ACM Symposium on Applied Computing (SAC 2008)*, Fortaleza, Cear a, Brazil, 16–20 March 2008. ACM. Special Track on Software Engineering.
- [139] Pablo Noriega and Carles Sierra. Electronic institutions: Future trends and challenges. In Matthias Klusch, Sascha Ossowski, and Onn Shehory, editors, *Cooperative Information Agents VI*, volume 2446 of *Lecture Notes in Computer Science*, pages 14–17. Springer, 2002. 6th International Workshop, CIA 2002, Madrid, Spain, September 18-20, 2002, Proceedings.
- [140] Quynh-Nhu Numi Tran and Graham C. Low. Comparison of ten agent-oriented methodologies. In Henderson-Sellers and Giorgini [87], chapter XII, pages 341–367.
- [141] Object Management Group. CORBA home page. <http://www.corba.org>.

- [142] Object Management Group. MOF home page. <http://www.omg.org/mof/>.
- [143] Object Management Group. UML home page. <http://www.uml.org>.
- [144] Andrea Omicini. **SODA**: Societies and infrastructures in the analysis and design of agent-based systems. In Paolo Ciancarini and Michael J. Wooldridge, editors, *Agent-Oriented Software Engineering*, volume 1957 of *LNCS*, pages 185–193. Springer, 2001. 1st International Workshop (AOSE 2000), Limerick, Ireland, 10 June 2000. Revised Papers.
- [145] Andrea Omicini. Towards a notion of agent coordination context. In Dan C. Marinescu and Craig Lee, editors, *Process Coordination and Ubiquitous Computing*, chapter 12, pages 187–200. CRC Press, Boca Raton, FL, USA, October 2002.
- [146] Andrea Omicini. Formal **ReSpecT** in the A&A perspective. *Electronic Notes in Theoretical Computer Sciences*, 175(2):97–117, June 2007. 5th International Workshop on Foundations of Coordination Languages and Software Architectures (FOCLASA’06), CONCUR’06, Bonn, Germany, 31 August 2006. Post-proceedings.
- [147] Andrea Omicini and Sascha Ossowski. Objective versus subjective coordination in the engineering of agent systems. In Matthias Klusch, Sonia Bergamaschi, Peter Edwards, and Paolo Petta, editors, *Intelligent Information Agents: An AgentLink Perspective*, volume 2586 of *LNAI: State-of-the-Art Survey*, pages 179–202. Springer-Verlag, March 2003.
- [148] Andrea Omicini, Sascha Ossowski, and Alessandro Ricci. Coordination infrastructures in the engineering of multiagent systems. In Bergenti et al. [8], chapter 14, pages 273–296.
- [149] Andrea Omicini and Alessandro Ricci. Reasoning about organisation: Shaping the infrastructure. *AI*IA Notizie*, XVI(2):7–16, June 2003.
- [150] Andrea Omicini, Alessandro Ricci, and Mirko Viroli. Formal specification and enactment of security policies through Agent Coordination Contexts. *Electronic Notes in Theoretical Computer Science*, 85(3):17–36, August 2003. 1st International Workshop “Security Issues in Coordination Models, Languages and Systems” (SecCo 2003), Eindhoven, The Netherlands, 28–29 June 2003. Proceedings.
- [151] Andrea Omicini, Alessandro Ricci, and Mirko Viroli. *Agens Faber*: Toward a theory of artifacts for MAS. *Electronic Notes in Theoretical Computer Sciences*, 2005. 1st International Workshop “Coordination and Organization” (CoOrg 2005), COORDINATION 2005, Namur, Belgium, 22 April 2005. Proceedings.
- [152] Andrea Omicini, Alessandro Ricci, and Mirko Viroli. *Agens Faber*: Toward a theory of artefacts for MAS. *Electronic Notes in Theoretical Computer Sciences*, 150(3):21–36, 29 May 2006. 1st International Workshop “Coordination and Organization” (CoOrg 2005), COORDINATION 2005, Namur, Belgium, 22 April 2005. Proceedings.
- [153] Andrea Omicini, Alessandro Ricci, and Mirko Viroli. Coordination artifacts as first-class abstractions for MAS engineering: State of the research. In Garcia [68], pages 71–90.

- [154] Andrea Omicini, Alessandro Ricci, Mirko Viroli, Marco Cioffi, and Giovanni Rimassa. Multi-agent infrastructures for objective and subjective coordination. *Applied Artificial Intelligence*, 18(9–10):815–831, October–December 2004. Special Issue: Best papers from EUMAS 2003: The 1st European Workshop on Multi-agent Systems.
- [155] Andrea Omicini and Giovanni Rimassa. Towards seamless agent middleware. In *IEEE 13th Inter. Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WET ICE 2004)*, pages 417–422, 2nd Inter. Workshop “Theory and Practice of Open Computational Systems” (TAPOCS 2004), Modena, Italy, 14–16 June 2004. IEEE CS.
- [156] Andrea Omicini and Giovanni Rimassa. Towards seamless agent middleware. In *IEEE 13th International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WET ICE 2004)*, pages 417–422, 2nd International Workshop “Theory and Practice of Open Computational Systems” (TAPOCS 2004), Modena, Italy, 14–16 June 2004. IEEE CS. Proceedings.
- [157] Andrea Omicini and Franco Zambonelli. Coordination for Internet application development. *Autonomous Agents and Multi-Agent Systems*, 2(3):251–269, September 1999.
- [158] Andrea Omicini and Franco Zambonelli. MAS as complex systems: A view on the role of declarative approaches. In João Alexandre Leite, Andrea Omicini, Leon Sterling, and Paolo Torroni, editors, *Declarative Agent Languages and Technologies*, volume 2990 of *LNAI*, pages 1–17. Springer, May 2004.
- [159] Andrea Omicini, Franco Zambonelli, Matthias Klusch, and Robert Tolksdorf, editors. *Coordination of Internet Agents: Models, Technologies, and Applications*. Springer-Verlag, March 2001.
- [160] OPEN Working Group. Open home page. <http://www.open.org.au/index.html>.
- [161] OpenConf. Home page. <http://www.zakongroup.com/technology/openconf.shtml>.
- [162] OSA. Home page. <http://osm7.cs.byu.edu/OSA/tutorial.html>.
- [163] Lin Padgham and Michael Winikof. Prometheus: A methodology for developing intelligent agents. In Fausto Giunchiglia, James Odell, and Gerhard Weiss, editors, *Agent-Oriented Software Engineering III*, volume 2585 of *LNCS*, pages 174–185. Springer, 2003. 3rd International Workshop (AOSE 2002), Bologna, Italy, 15 July 2002. Revised Papers and Invited Contributions.
- [164] Lin Padgham and Michael Winikoff. Prometheus: A practical agent oriented methodology. In Henderson-Sellers and Giorgini [87], chapter V, pages 107–135.
- [165] George A. Papadopoulos. Models and technologies for the coordination of internet agents: A survey. In Omicini et al. [159], pages 25–56.
- [166] George A. Papadopoulos and Farhad Arbab. Coordination models and languages. *Advances in Computers*, 46:330–401, 1998.

- [167] H. Van Dyke Parunak. “go to the ant”: Engineering principles from natural agent systems. *Annals of Operation Research*, 75:69–101, 1997.
- [168] PASSI Group. PASSI home page. <http://mozart.csai.unipa.it/passi/>.
- [169] Juan Pavón and Jorge J. Gómez-Sanz. Agent oriented software engineering with ingenias. In Vladimír Marík, Jörg P. Müller, and Michal Pechoucek, editors, *CEEMAS*, volume 2691 of *LNCS*, pages 394–403. Springer, 2003. 3rd International Central and Eastern European Conference on Multi-Agent Systems, CEEMAS 2003, Prague, Czech Republic, June 16-18, 2003, Proceedings.
- [170] Juan Pavón, Jorge J. Gómez-Sanz, and Rubén Fuentes. The INGENIAS methodology and tools. In Henderson-Sellers and Giorgini [87], chapter IX, pages 236–276.
- [171] Mor Peleg and Dov Dori. The model multiplicity problem: Experimenting with real-time specification methods. *IEEE Transactions on Software Engineering*, 26(8):742–759, 2000.
- [172] Loris Penserini, Anna Perini, Angelo Susi, and John Mylopoulos. From stakeholder intentions to software agent implementations. In Eric Dubois and Klaus Pohl, editors, *Advanced Information Systems Engineering*, volume 4001 of *Lecture Notes in Computer Science*, pages 465–479. Springer, 2006. 18th International Conference, CAiSE 2006, Luxembourg, Luxembourg, June 5-9, 2006, Proceedings.
- [173] Gauthier Picard, Carole Bernon, and Marie-Pierre Gleizes. Cooperative agent model within ADELFE framework: An application to a timetabling problem. In Nicholas R. Jennings, Carles Sierra, Liz Sonenberg, and Milind Tambe, editors, *Proceedings of the 3rd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2004)*, volume 3, pages 1506–1507, New York, USA, 19–23 July 2004. ACM Press.
- [174] Jolita Ralyté and Colette Rolland. An approach for method reengineering. In *Conceptual Modeling*, pages 471–484, London, UK, 2001. Springer-Verlag. 20th International Conference (ER 2001), Yokohama, Japan, 27-30 November 2001. Proceedings.
- [175] RBAC. Role Base Access Control- home page. <http://csrc.nist.gov/rbac/>.
- [176] Alessandro Ricci. *Engineering Agent Societies with Coordination Artifacts and Supporting Infrastructures*. PhD thesis, march 2008.
- [177] Alessandro Ricci, Mirko Viroli, and Andrea Omicini. An RBAC approach for securing access control in a MAS coordination infrastructure. In Mike Barley and et al., editors, *1st International Workshop Safety and Security in MultiAgent Systems*, pages 110–124, 2004.
- [178] Alessandro Ricci, Mirko Viroli, and Andrea Omicini. Programming MAS with artifacts. In Rafael P. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah Seghrouchni, editors, *Programming Multi-Agent Systems*, volume 3862 of *LNAI*, pages 206–221. Springer, March 2006. 3rd Inter. Workshop (PROMAS 2005), AAMAS 2005, The Netherlands, Revised and Invited Papers.

- [179] Alessandro Ricci, Mirko Viroli, and Andrea Omicini. **CArtAgO**: An infrastructure for engineering computational environments in MAS. In Danny Weyns, H. Van Dyke Parunak, and Fabien Michel, editors, *3rd Inter. Workshop "Environments for Multi-Agent Systems" (E4MAS 2006)*, pages 102–119, 8 May 2006.
- [180] Alessandro Ricci, Mirko Viroli, and Andrea Omicini. "Give agents their artifacts": The A&A approach for engineering working environments in MAS. In Edmund Durfee, Makoto Yokoo, Michael Huhns, and Onn Shehory, editors, *6th International Joint Conference "Autonomous Agents & Multi-Agent Systems" (AAMAS 2007)*, pages 601–603, Honolulu, Hawai'i, USA, 14–18 May 2007. IFAAMAS.
- [181] Alessandro Ricci, Mirko Viroli, and Andrea Omicini. **CArtAgO**: A framework for prototyping artifact-based environments in MAS. In Danny Weyns, H. Van Dyke Parunak, and Fabien Michel, editors, *Environments for MultiAgent Systems*, volume 4389 of *LNAI*, pages 67–86. Springer, February 2007. 3rd International Workshop (E4MAS 2006). Selected Revised and Invited Papers.
- [182] Diana Richards, Brendan D. McKay, and Whitman A. Richards. Collective choice and mutual knowledge structures. *Advances in Complex Systems*, 1:221–236, jun & sep 1998.
- [183] Giovanni Rimassa. *Runtime support for distributed multi-agent systems*. PhD thesis, University of Parma, 2003.
- [184] Luis M Rocha. Complex systems modeling: Using metaphors from nature in simulation and scientific models. <http://www.informatics.indiana.edu/rocha/complex/csm.html>, 1999.
- [185] Colette Rolland. A comprehensive view of process engineering. In Barbara Pernici and Costantino Thanos, editors, *CAiSE*, volume 1413 of *Lecture Notes in Computer Science*, pages 1–24. Springer, 1998. Advanced Information Systems Engineering, 10th International Conference CAiSE'98, Pisa, Italy, June 8-12, 1998, Proceedings.
- [186] Colette Rolland, Naveen Prakash, and A. Benjamen. A multi-model view of process modelling. *Requirement Engineering*, 4(4):169–187, 1999.
- [187] Davide Rossi and Fabio Vitali. Internet-based coordination environments and document-based applications: A case study. In Ciancarini and Wolf [31], pages 259–274.
- [188] Rossella Rubino, Ambra Molesini, and Enrico Denti. Owl-s for describing artifacts. In Andrea Omicini, Barbara Dunin-Keplicz, and Julian Padget, editors, *4th European Workshop on Multi-Agent Systems (EUMAS 2006)*, number 223 in CEUR Workshop Proceedings, pages 195–206. Sun SITE Central Europe, RWTH Aachen University, 14-15 December 2006. Fourth European Workshop on Multi-Agent Systems Lisbon, Portugal.
- [189] James E. Rumbaugh, Michael R. Blaha, William J. Premerlani, Frederick Eddy, and William E. Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.

- [190] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.
- [191] Vladimiro Sassone. Management of electronic submission, refereeing, and PC meeting. <http://www.logic.at/staff/salzer/confpack/README.icalp>, 1996.
- [192] John A. Sauter, Robert S. Matthews, H. Van Dyke Parunak, and Sven Brueckner. Performance of digital pheromones for swarming vehicle control. In Dignum et al. [51], pages 903–910.
- [193] Kjeld Schmidt and Carla Simone. Coordination mechanisms: Towards a conceptual foundation of CSCW systems design. *Computer Supported Cooperative Work*, 5(2/3):155–200, 1996.
- [194] Adriano Scutellà. Simulation of conference management using an even-driven coordination language. In Ciancarini and Wolf [31], pages 243–258.
- [195] Lijun Shan and Hong Zhu. Camle: A caste-centric agent-oriented modelling language and environment. In Ricardo Choren, Alessandro F. Garcia, Carlos José Pereira de Lucena, and Alexander B. Romanovsky, editors, *SELMAS*, volume 3390 of *LNCS*, pages 144–161. Springer, 2005. Software Engineering for Multi-Agent Systems III, Research Issues and Practical Applications [the book is a result of SELMAS 2004].
- [196] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, May 1996.
- [197] Herbert A. Simon. *The Sciences of the Artificial*. The MIT Press, 3rd edition, 1996.
- [198] Olivier Simonin and Franck Gechter. An environment-based methodology to design reactive multi-agent systems for problem solving. In Weyns et al. [221], pages 32–49. 2nd International Workshop Environments for Multi-Agent Systems 2005, Utrecht, The Netherlands, July 25, 2005, Selected Revised and Invited Papers.
- [199] Ian Sommerville. *Software Engineering 8th Edition*. Addison-Wesley, 2007.
- [200] SPEM. SPEM Software Process Engineering Meta-Model home page. <http://www.omg.org/technology/documents/formal/spem.htm>.
- [201] Arnon Sturm, Dov Dori, and Onn Shehory. Single-model method for specifying multi-agent systems. In *AAMAS '03: Proceedings of the Second International joint conference on Autonomous Agents and Multiagent Systems*, pages 121–128. ACM Press, 2003.
- [202] Arnon Sturm and Onn Shehory. A comparative evaluation of agent-oriented methodologies. In Bergenti et al. [8], chapter 7, pages 127–149.
- [203] Arnon Sturm and Onn Shehory. A framework for evaluating agent-oriented methodologies. In Paolo Giorgini, Brian Henderson-Sellers, and Michael Winikoff, editors, *Agent-Oriented Information Systems*, volume 3030 of *LNCS*, pages 94–109. Springer, 24 June 2004. 5th

- International Bi-Conference Workshop, AOIS 2003, Melbourne, Australia, July 14, 2003 and Chicago, USA, October 13, 2003, Revised Selected Papers.
- [204] Sun Microsystems. J2EE home page. <http://java.sun.com/j2ee>.
- [205] Katia P. Sycara, Massimo Paolucci, Martin Van Velsen, and Joseph A. Giampapa. The RETSINA MAS infrastructure. *Autonomous Agents and Multi-Agent Systems*, 7(1-2):29–48, 2003.
- [206] M. Tambe, D. Pynadath, N. Chauvat, A. Das, and G. Kaminka. Adaptive agent integration architectures for heterogeneous team members. In *International Conference on Multi-Agent Systems*, pages 301–308. IEEE Computer Society, 2000. 4th International Conference on Multi-Agent Systems (ICMAS 2000), 10-12 July 2000, Boston, MA, USA.
- [207] Milind Tambe and Weixiong Zhang. Towards flexible teamwork in persistent teams: Extended report. *Autonomous Agents and Multi-Agent Systems*, 3(2):159–183, 2000.
- [208] Kuldar Taveret and Gerd Wagner. Towards radical agent-oriented software engineering processes based on AOR modelling. In Henderson-Sellers and Giorgini [87], chapter X, pages 277–316.
- [209] TEAMCORE Group. TEAMCORE home page. <http://teamcore.usc.edu/>.
- [210] Quynh-Nhu Numi Tran, Graham Low, and Mary-Anne Williams. A preliminary comparative feature analysis of multi-agent systems development methodologies. In Paolo Bresciani, Paolo Giorgini, Brian Henderson-Sellers, Graham Low, and Michael Winikoff, editors, *Agent-Oriented Information Systems II*, volume 3508 of *Lecture Notes in Computer Science*, pages 157–168. Springer, 2005. 6th International Bi-Conference Workshop, AOIS 2004, Riga, Latvia, June 8, 2004 and New York, NY, USA, July 20, 2004, Revised Selected Papers.
- [211] Tropos Group. Tropos home page. <http://www.troposproject.org/>.
- [212] TuCSon at SourceForge. <http://tucson.sourceforge.net>.
- [213] Mirko Viroli, Tom Holvoet, Alessandro Ricci, Kurt Schelfhout, and Franco Zambonelli. Infrastructures for the environment of multiagent systems. In *Autonomous Agents and Multi-Agent Systems* [220], pages 49–60.
- [214] Mirko Viroli and Andrea Omicini. Coordination as a service: Ontological and formal foundation. *Electronic Notes in Theoretical Computer Science*, 68(3):457–482, March 2003. 1st International Workshop “Foundations of Coordination Languages and Software Architecture” (FOCLASA 2002), Brno, Czech Republic, 24 August 2002. Proceedings.
- [215] Mirko Viroli, Andrea Omicini, and Alessandro Ricci. Engineering MAS environment with artifacts. In Danny Weyns, H. Van Dyke Parunak, and Fabien Michel, editors, *2nd International Workshop “Environments for Multi-Agent Systems” (E4MAS 2005)*, pages 62–77, AAMAS 2005, Utrecht, NL, 26 July 2005.

- [216] Mirko Viroli, Alessandro Ricci, and Andrea Omicini. Operating instructions for intelligent agent coordination. *The Knowledge Engineering Review*, 21(1):49–69, March 2006.
- [217] Greg Wagner. The agent-object relationship metamodel: towards a unified view of state and behaviour. *Information Systems*, 28(5):475–504, July 2003.
- [218] Webchairing. Home page. <http://www.webchairing.com/webchairing/index-eng.asp>.
- [219] Danny Weyns, Andrea Omicini, and James Odell. Environment as a first-class abstraction in multi-agent systems. In *Autonomous Agents and Multi-Agent Systems* [220], pages 5–30.
- [220] Danny Weyns and H. Van Dyke Parunak. Special issue on environments for multi-agent systems. *Autonomous Agents and Multi-Agent Systems*, 14(1):1–116, February 2007.
- [221] Danny Weyns, H. Van Dyke Parunak, and Fabien Michel, editors. *Environments for Multi-Agent Systems II*, volume 3830 of *Lecture Notes in Computer Science*. Springer, 2006. 2nd International Workshop Environments for Multi-Agent Systems 2005, Utrecht, The Netherlands, July 25, 2005, Selected Revised and Invited Papers.
- [222] Danny Weyns, Kurt Schelfhout, Tom Holvoet, and Tom Lefever. Decentralized control of E'GV transportation systems. In Dignum et al. [51], pages 67–74.
- [223] Danny Weyns, Giuseppe Vizzari, and Tom Holvet. Environments for situated multi-agent systems: Beyond infrastructure. In Weyns et al. [221], pages 1–17. 2nd International Workshop Environments for Multi-Agent Systems 2005, Utrecht, The Netherlands, July 25, 2005, Selected Revised and Invited Papers.
- [224] Mark F. Wood and Scott A. DeLoach. An overview of the multiagent systems engineering methodology. In Paolo Ciancarini and Michael J. Wooldridge, editors, *Agent-Oriented Software Engineering*, volume 1957 of *LNCS*, pages 207–221. Springer-Verlag, 2001. 1st International Workshop (AOSE 2000), Limerick, Ireland, 10 June 2000. Revised Papers.
- [225] M. Wooldridge, N. R. Jennings, and D. Kinny. The gaia methodology for agent-oriented analysis and design. *Autonomous Agents and Multi-Agent Systems*, 3(3):285–312, September 2000.
- [226] Michael Wooldridge and Paolo Ciancarini. Agent-oriented software engineering: The state of the art. In Paolo Ciancarini and Michael Wooldridge, editors, *AOSE*, volume 1957 of *LNCS*, pages 1–28. Springer, 2000. First International Workshop, AOSE 2000, Limerick, Ireland, June 10, 2000, Revised Papers.
- [227] Eric Yu. *Modelling Strategic Relationships for Process Reengineering*. PhD thesis, University of Toronto, 1995.
- [228] Franco Zambonelli, Nicholas Jennings, and Michael Wooldridge. Multiagent systems as computational organizations: the Gaia methodology. In Henderson-Sellers and Giorgini [87], chapter VI, pages 136–171.

-
- [229] Franco Zambonelli, Nicholas R. Jennings, and Michael Wooldridge. Developing multi-agent systems: The Gaia methodology. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 12(3):317–370, July 2003.
- [230] Franco Zambonelli and Andrea Omicini. Challenges and research directions in agent-oriented software engineering. *Autonomous Agents and Multi-Agent Systems*, 9(3):253–283, November 2004. Special Issue: Challenges for Agent-Based Computing.
- [231] Franco Zambonelli and H. Van Dyke Parunak. Towards a paradigm change in computer science and software engineering: a synthesis. *The Knowledge Engineering Review*, 18(4):329–342, 2003.